



# Lecția 10:

## Procesorul:

### Tehnica de pipeline - II

G Ștefănescu — Universitatea București

Arhitectura sistemelor de calcul, Sem.1

Octombrie 2016—Februarie 2017

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



# Procesorul: Tehnica de pipeline

---

## Cuprins:

- Tehnica de pipeline
- Calea de date cu pipeline
- Controlul pentru implementari cu pipeline
- *Hazard de date si avansari*
- Hazard de date si stationari
- Hazard la ramificatii
- Exceptii
- Concluzii, diverse, etc.



# Dependenta de date

## Dependenta de date vs. pipeline:

- Potențialul extraordinar oferit de tehnica de pipeline este atenuat de dificultatea de a trata eficient excepțiile produse de diversele tipuri de *hazard*, care întârzie procesarea.
- Aici analizăm hazardul creat de *interdependența datelor*.
- In exemplul de mai jos, ultimele 4 instrucțiuni sunt depedente de prima prin intermediul registrului \$2.

sub \$2, \$1, \$3; (1)

and \$12, \$2, \$5; (2)

or \$13, \$6, \$2; (3)

add \$14, \$2, \$2; (4)

sw \$15, 100(\$2); (5)

# ..Dependenta de date

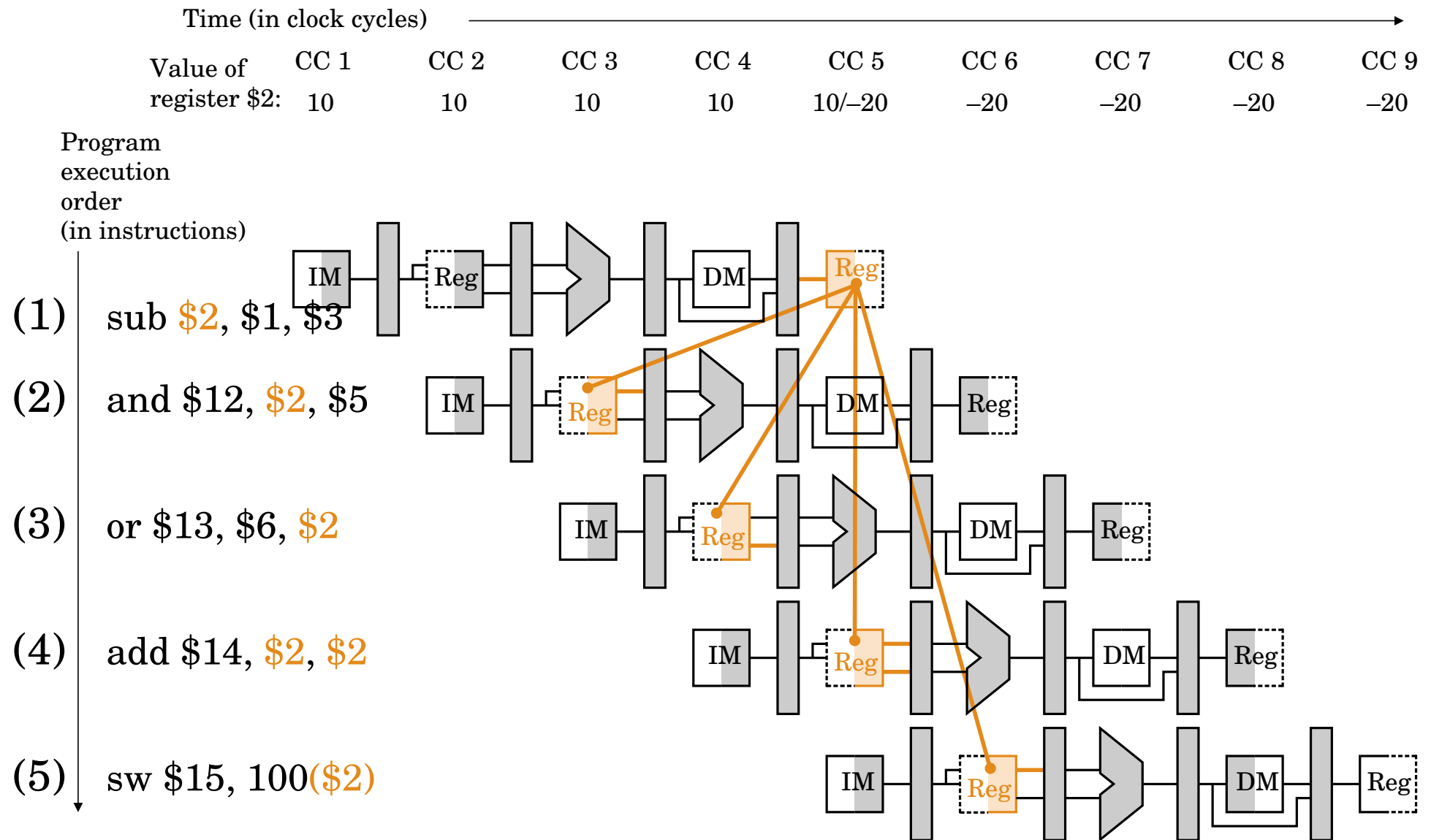


Figura ilustrează *dependențele de date* între 5 instrucțiuni procesate cu *pipeline*; ilustrarea este schematică spre a reliefa dependențele.



# ..Dependenta de date

## Dependenta de date vs. pipeline:

- Dependența de date este ilustrată în figura de mai sus.
- Figura descrie cazul când \$2 conține valoarea 10 înainte de scădere și -20 după.
- Care sunt hazardurile care apar?
  - pentru perechile  $(1,2)$  și  $(1,3)$  *avem* hazarduri;
  - pentru perechea  $(1,4)$  *nu avem hazard*, dacă se aplică convenția de a avea scriere în prima jumătate de ceas și citire în a doua;
  - pentru perechea  $(1,5)$  *nu avem* hazard.



# ..Dependenta de date

## Interfata hardware-software:

- O soluție facilă la hazardurile de mai sus este de a folosi o instrucțiune specială **nop** care nu face nimic.
- Compilatorul poate genera secvența echivalentă

```
sub $2, $1, $3;                (1)
```

```
nop
```

```
nop
```

```
and $12, $2, $5;              (2)
```

```
or  $13, $6, $2;              (3)
```

```
add $14, $2, $2;              (4)
```

```
sw  $15, 100($2);             (5)
```

care este corectă, dar inserează o întârziere de 2 cicluri.

- Rămân de căutat soluții mai performante.

## Avansarea:

- Putem folosi tehnica de *avansare (forwarding)* pentru a soluționa hazardurile de date.
- *Sursa:* Noile valori produse de instrucțiuni pentru registrul destinație pot proveni:
  - din ALU (instrucțiune de format-R) și, înainte de finalizare, se găsesc în regiștrii pipeline EX/MEM și MEM/WB;
  - din memorie (instrucțiune load) și, înainte de finalizare, se găsesc în registrul pipeline MEM/WB.
- *Destinația:* Pentru un hazard soluționat prin avansare, utilizarea acestor valori se face
  - în faza de execuție, folosindu-se pentru unul din argumentele ALU.

## Avansarea (cont.)

- Tipurile de *hazard* care pot fi *rezolvate* cu tehnica de avansare se pot clasifica astfel:
  - *1a*:  $\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}$
  - *1b*:  $\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRt}$
  - *2a*:  $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs}$
  - *2b*:  $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}$

Pe scurt, regiștri  $rs, rt$  pentru ALU provin din EX/MEM la tipul 1 și din MEM/WB la tipul 2.

- In exemplul anterior, perechea (1,2) este hazard de tip 1a, iar perechea (1,3) este de tip 2b.



# ..Avansarea

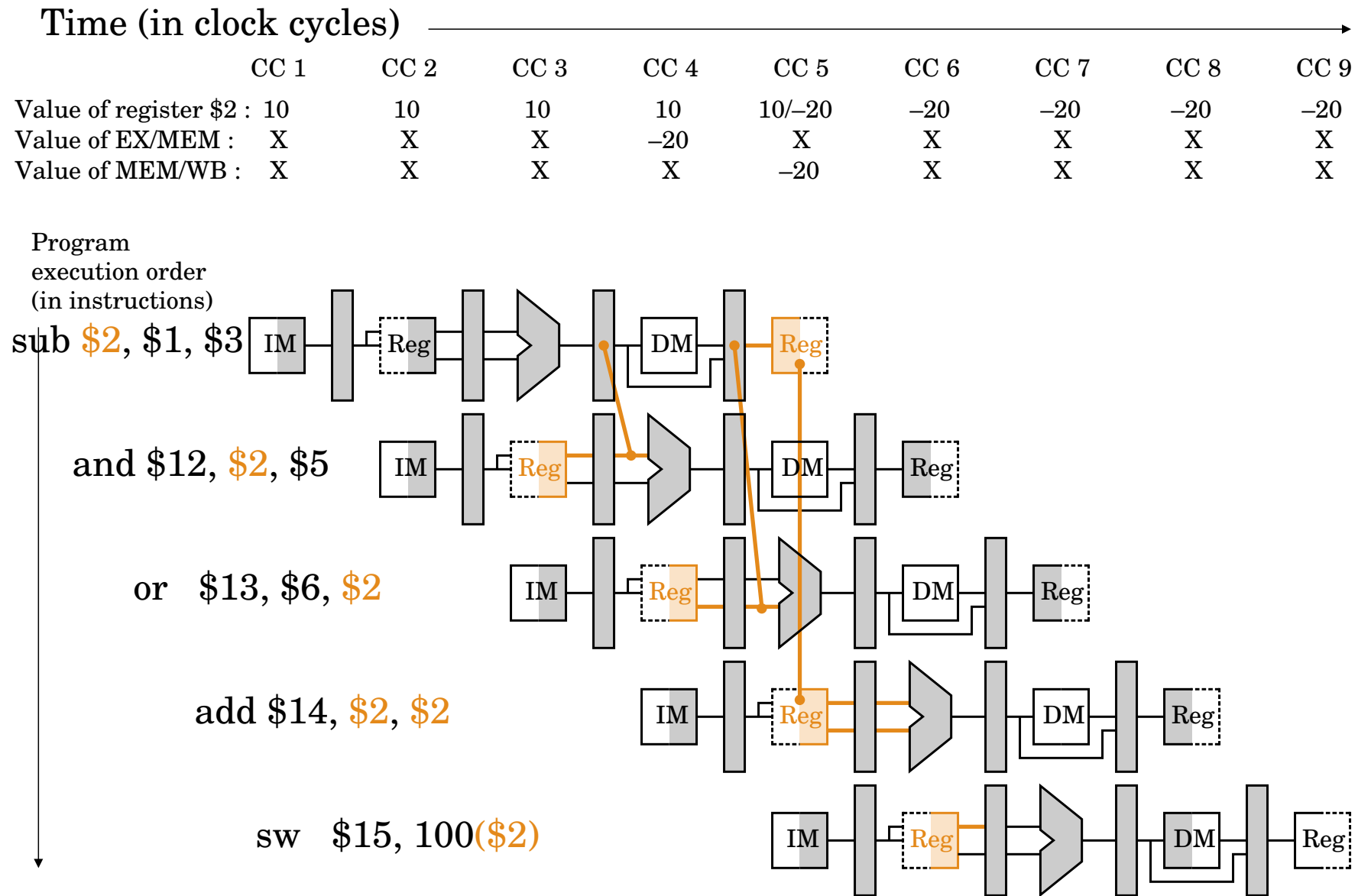


Figura ilustrează tehnica de *avansare* pentru hazardurile de date: luăm datele necesare *înainte de finalizarea* procesării instrucțiunii.



# ..Avansarea

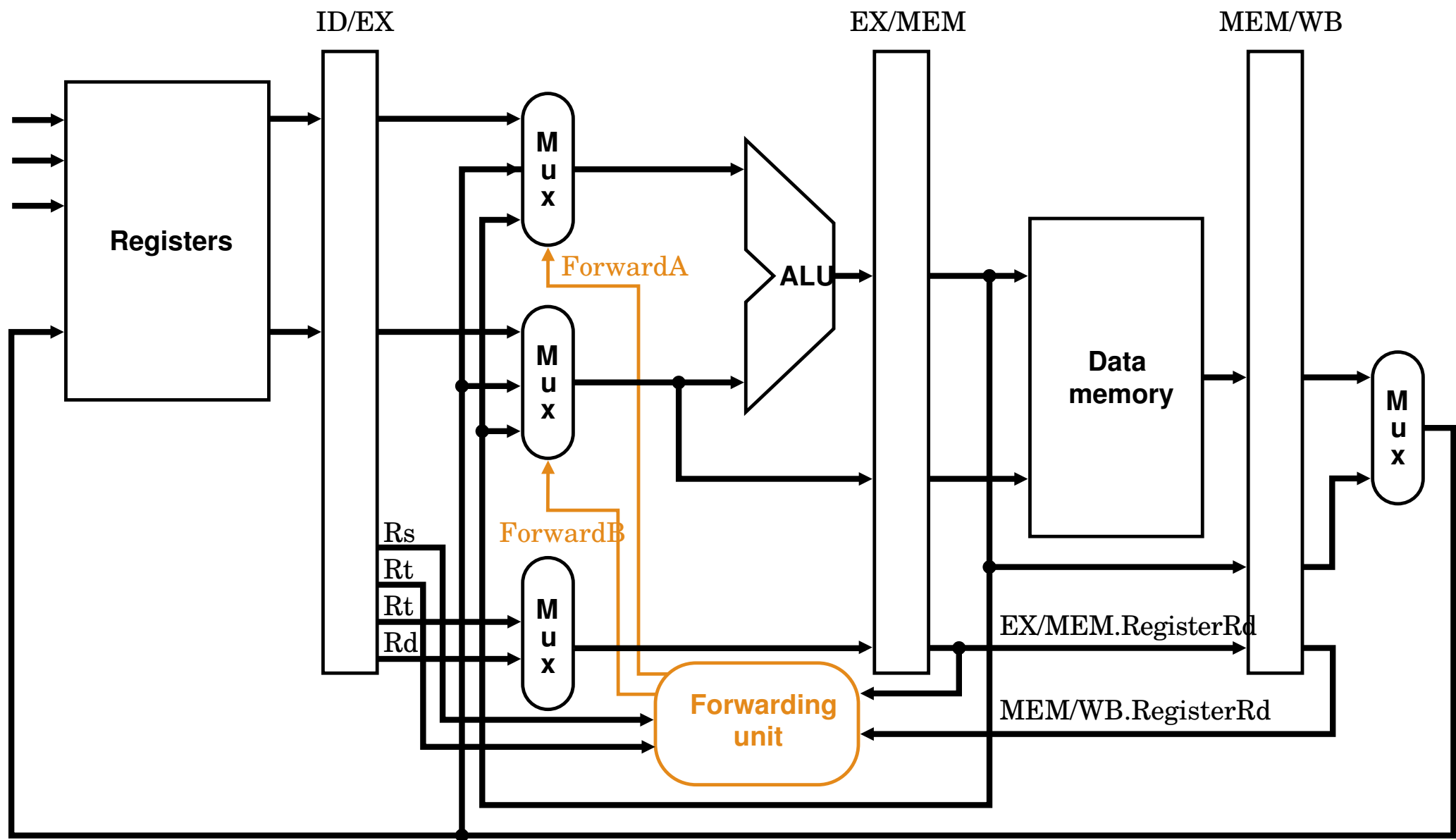
## Avansarea (cont.)

Condiții pentru *detectarea hazardului*:

- Pentru un hazard de *tip 1*, condițiile sunt:
  - semnalul EX/MEM.RegWrite este setat (i.e., instrucțiunea curentă scrie într-un registru);
  - EX/MEM.RegisterRd  $\neq$  0 (i.e., registrul destinație pentru avansare nu este registrul 0, care nu poate fi modificat);
- Similar pentru un hazard de *tip 2*, dar folosim registrul pipeline MEM/WB.

Condiții pentru *avansarea datelor*

- Folosim 2 semnale noi de control pentru avansarea datelor ForwardA și ForwardB - vezi figura.



In roșu sunt prezentate noile *semnalele de control* necesare pentru rezolvarea hazardului de date cu tehnica de *avansare* (pe o cale de date simplificată).



# ..Dependenta de date

Condițiile de *detectare hazard* și rezolvare cu *avansări*:

- *Tip 1: în faza EX* (execuție):
  - $EX/MEM.RegWrite \wedge EX/MEM.RegisterRd \neq 0$   
 $\wedge EX/MEM.RegisterRd = ID/EX.RegisterRs \Rightarrow ForwardA = 10$
  - $EX/MEM.RegWrite \wedge EX/MEM.RegisterRd \neq 0$   
 $\wedge EX/MEM.RegisterRd = ID/EX.RegisterRt \Rightarrow ForwardB = 10$
- *Tip 2: în faza MEM* (access memorie) - *provizoriu*:
  - $MEM/WB.RegWrite \wedge MEM/WB.RegisterRd \neq 0$   
 $\wedge MEM/WB.RegisterRd = ID/EX.RegisterRs \Rightarrow ForwardA = 01$
  - $MEM/WB.RegWrite \wedge MEM/WB.RegisterRd \neq 0$   
 $\wedge MEM/WB.RegisterRd = ID/EX.RegisterRt \Rightarrow ForwardB = 01$

## ..Dependenta de date

- Condițiile de mai sus *nu* sunt *disjuncte*; e.g., ele se pot suprapune, ca în cazul

```
add $1,$1,$2;
```

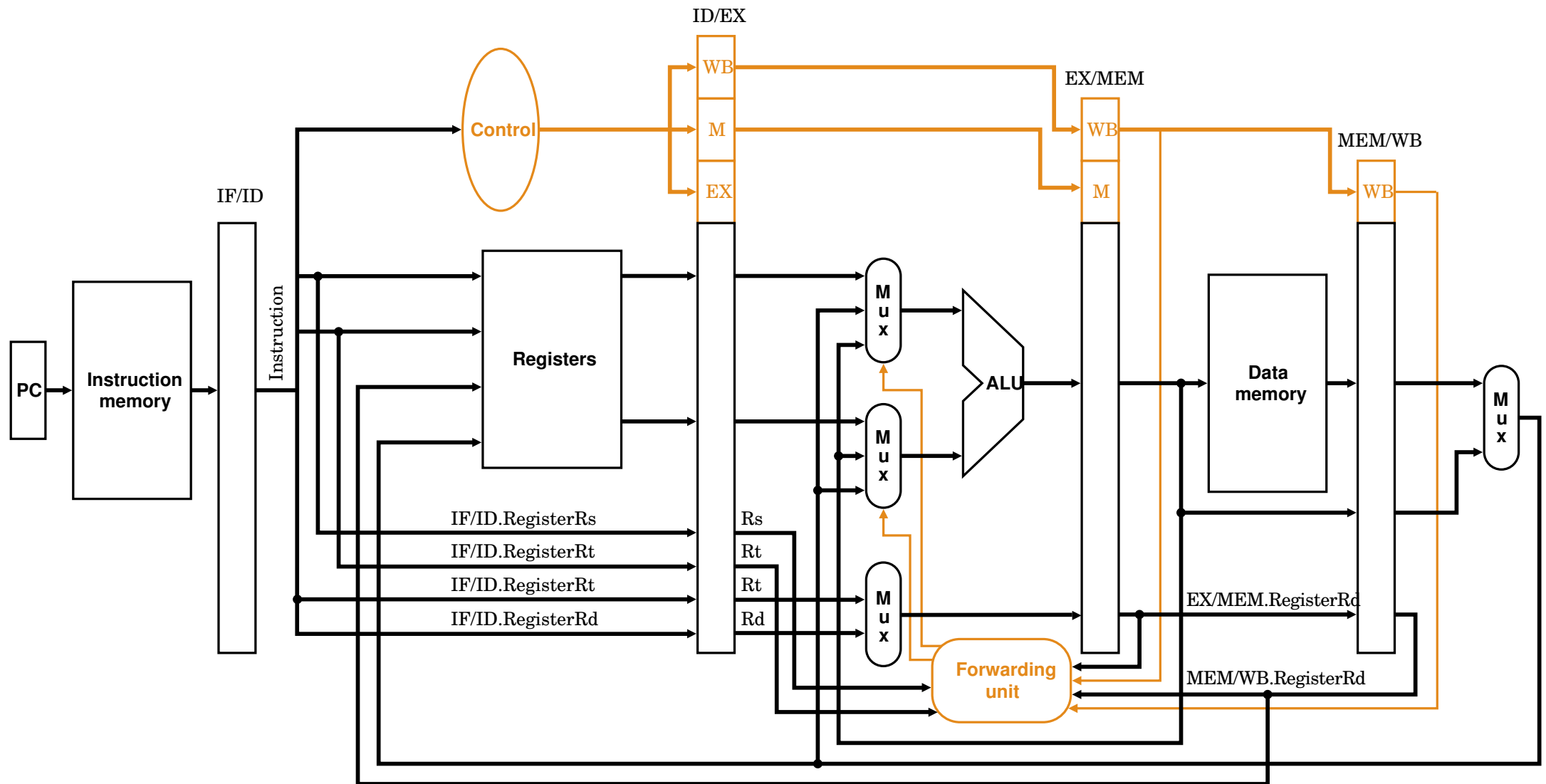
```
add $1,$1,$3;
```

```
add $1,$1,$4;
```

Hazardul se rezolvă luând cel mai recent rezultat (din faza EX, nu MEM). Deci apare o condiție suplimentară pentru Tipul 2.

- *Tip 2: în faza MEM* (access memorie) - *final*:
  - $\text{MEM/WB.RegWrite} \wedge \text{MEM/WB.RegisterRd} \neq 0$   
     $\wedge \text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRs}$   
     $\wedge \text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs} \Rightarrow \text{ForwardA} = 01$
  - $\text{MEM/WB.RegWrite} \wedge \text{MEM/WB.RegisterRd} \neq 0$   
     $\wedge \text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRt}$   
     $\wedge \text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt} \Rightarrow \text{ForwardB} = 01$

# ..Dependenta de date



Schiță a căii de date cu *componenta de control pentru avansare* utilizată pentru rezolvarea hazardului de date (ilustrată pe o cale de date simplificată).



- In exemplele de mai jos ilustrăm cum funcționează tehnica de avansare pe următoarea secvență de program

$$\text{add} \quad \$9, \$4, \$2; \quad (d)$$

- $k_a$  valoarea registrului  $k$  *la intrarea* în instrucțiunea  $a$ .
- $k_{+a}$  valoarea registrului  $k$  *la ieșirea* din instrucțiunea  $a$ .
- $k_{<a}$  o valoarea nedefinită a registrului  $k$  *înainte* de instrucțiunea  $a$ .



# ..Dependenta de date

---

## Exemplu (cont.)

- Condițiile de corectitudine sunt:

$$1_a - 3_a = 2_{+a} = 2_b = 2_c = 2_d;$$

$$2_b \wedge 5_b = 4_{+b} = 4_c;$$

$$4_c \vee 2_c = 4_{+c} = 4_d$$

- Tabelul de mai jos indică evoluția calculului, unde:
  - o linie conține toate valorile accesibile la un ceas dat;
  - culorile indică de unde se iau valorile corecte pentru regiștri ALU: albastru - din EX/MEM; roșu - din MEM/WB;



# ..Dependenta de date

## Exemplu (cont.)

- Tabel cu evoluția regiștrilor semnificativi:

Ceas	PC	IF/ID	ID/EX(rs,rt)	EX/MEM	MEM/WB
1	$a$				
2	$b$	$I_a$			
3	$c$	$I_b$	$1_a, 3_a$		
4	$d$	$I_c$	$2_{<b}, 5_b$	$1_a - 3_a$	
5		$I_d$	$4_{<c}, 2_{<c}$	$2_b \wedge 5_b$	$1_a - 3_a$
6			$4_{<d}, 2_d$	$4_c \vee 2_c$	$2_b \wedge 5_b$
7				$4_d + 2_d$	$4_c \vee 2_c$
8					$4_d + 2_d$

- De notat că în pasul 6, registrul 4 de folosit este cel din EX/MEM, nu cel din MEM/WB.

*(Ciclul 3)*

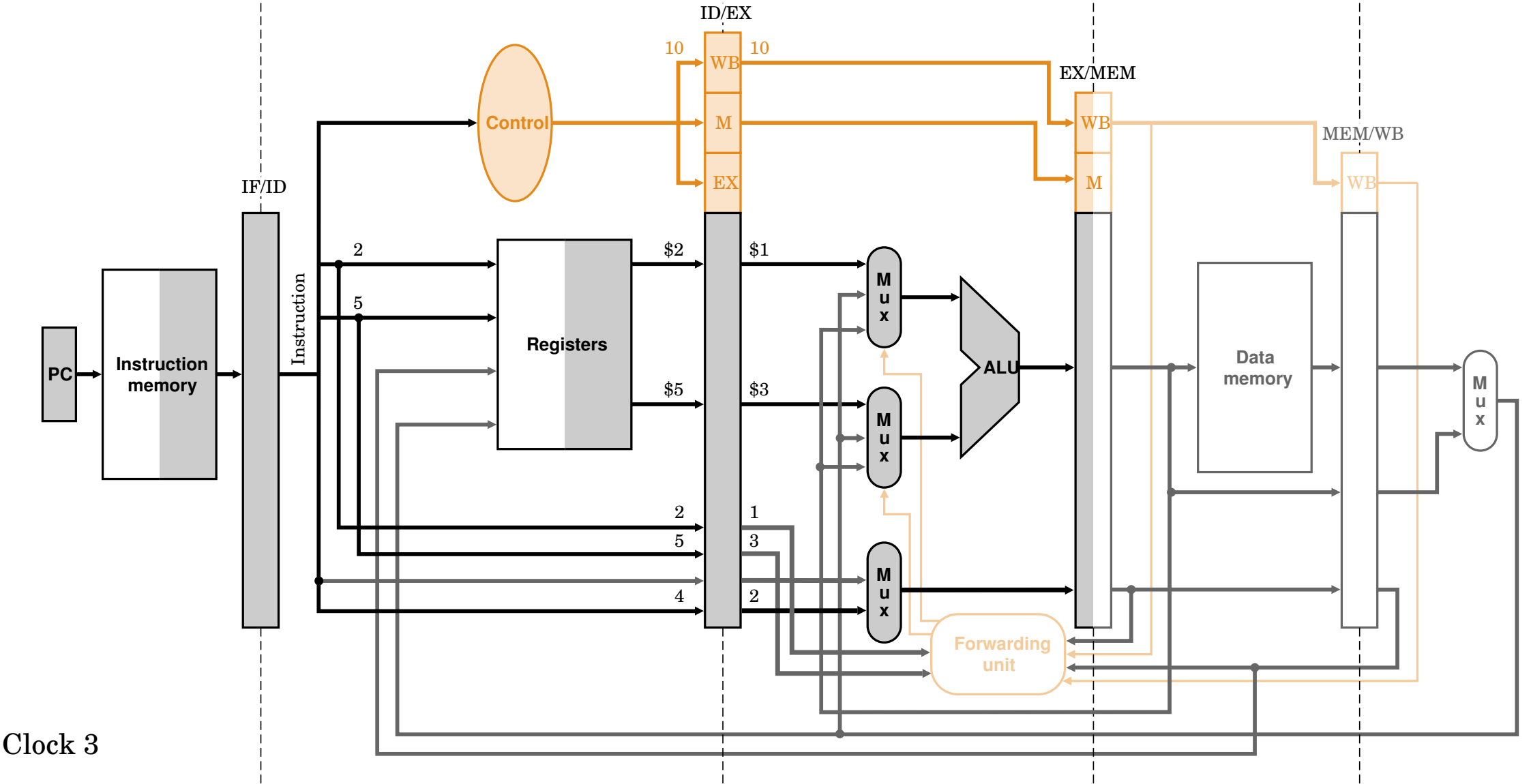
or \$4, \$4, \$2

and \$4, \$2, \$5

sub \$2, \$1, \$3

before<1>

before<2>



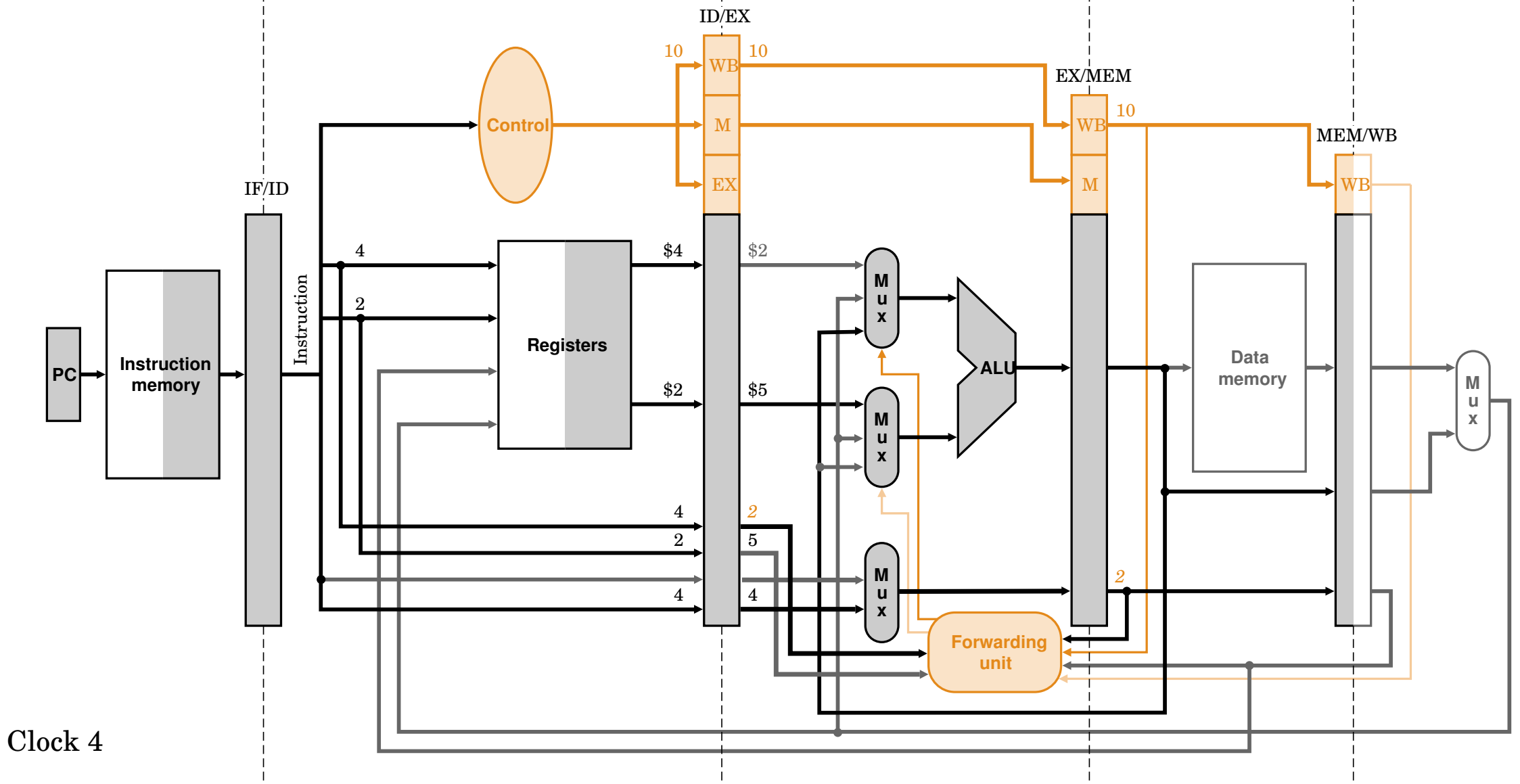
add \$9, \$4, \$2

or \$4, \$4, \$2

and \$4, \$2, \$5

sub \$2, ...

before<1>



Clock 4

*(Ciclul 4)*

*(Ciclul 5)*

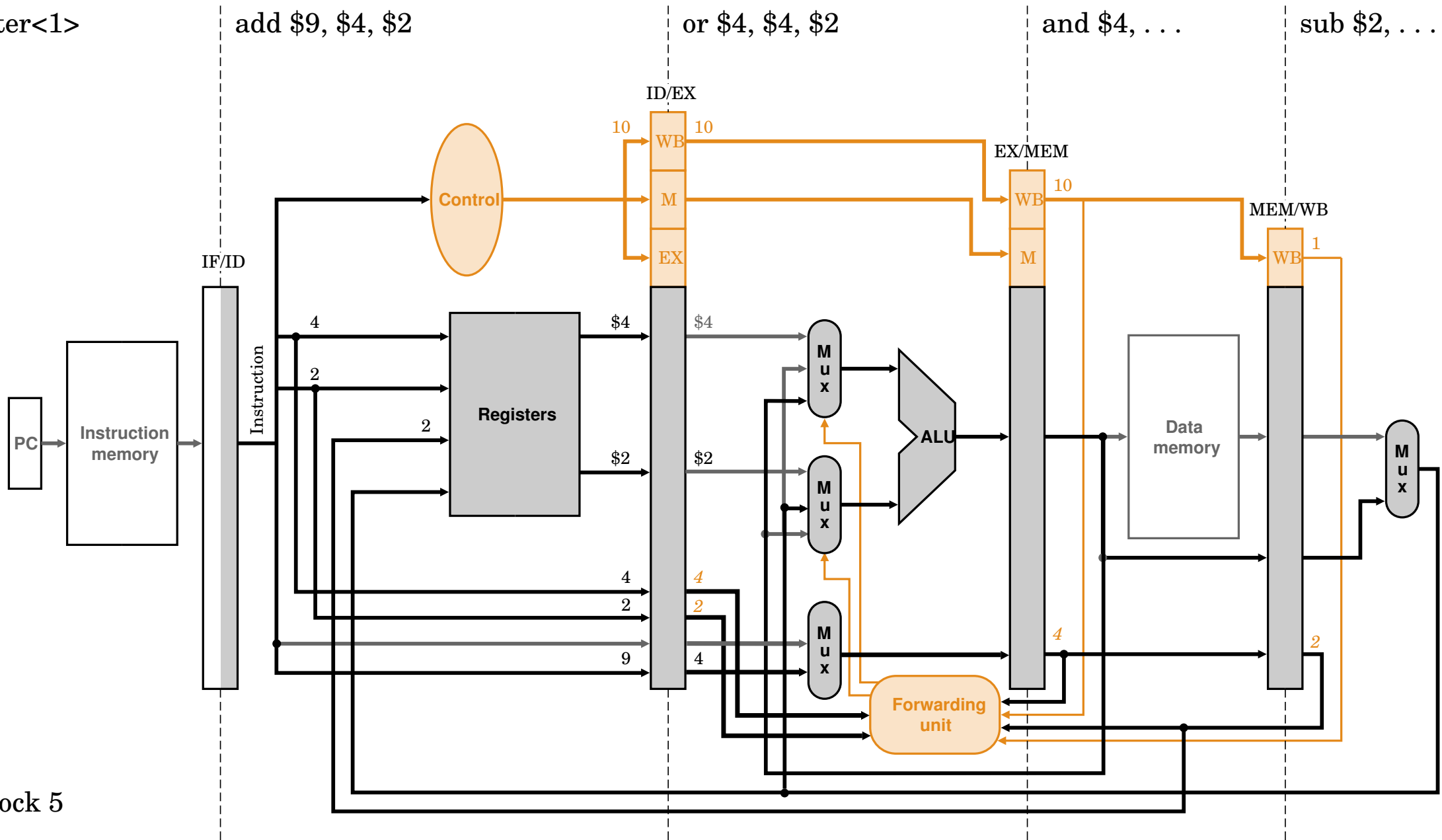
after<1>

add \$9, \$4, \$2

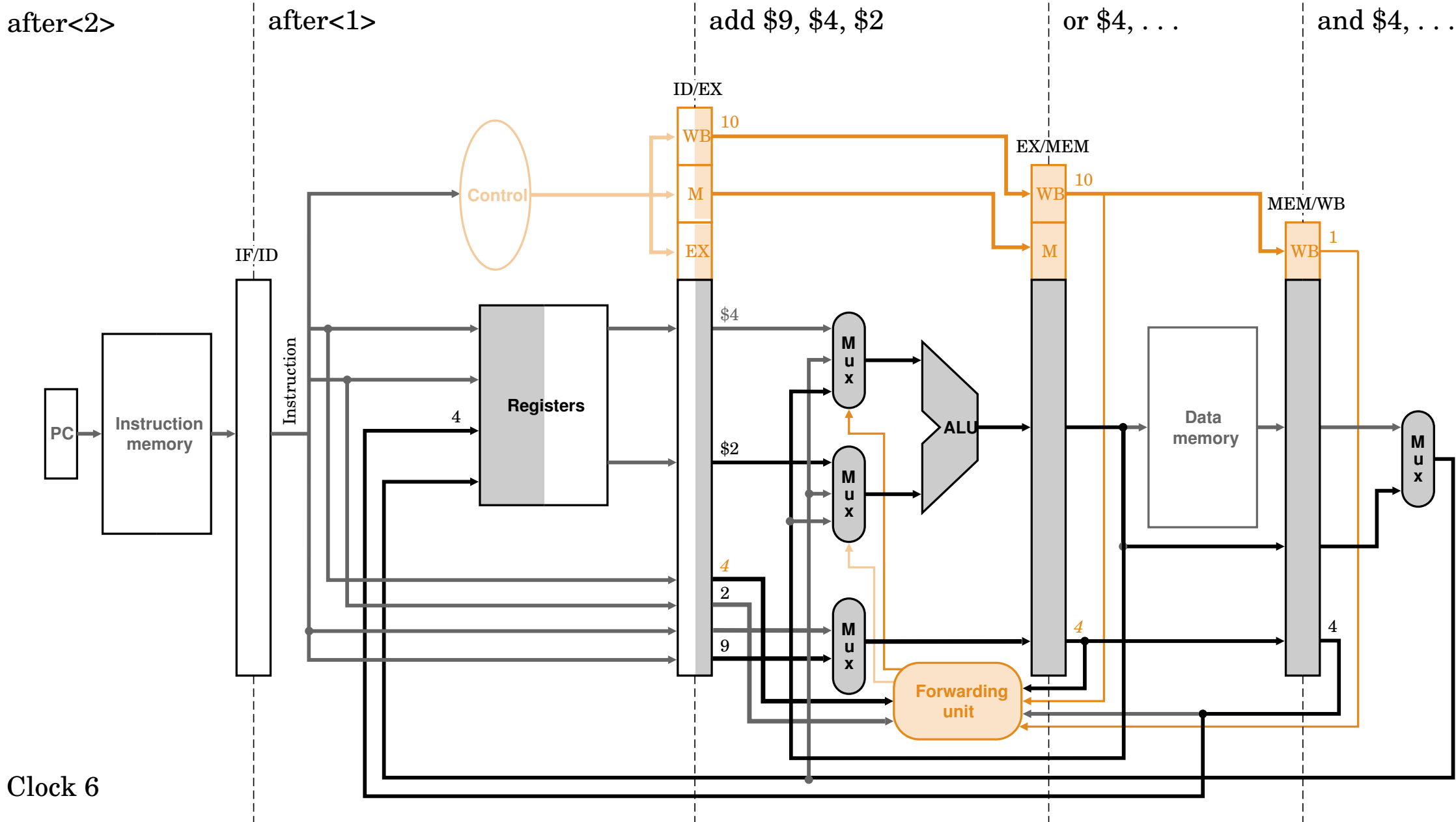
or \$4, \$4, \$2

and \$4, \dots\$

sub \$2, . . .



## Clock 5



Clock 6

*(Ciclul 6)*



# ..Dependenta de date

## Comentarii:

- Analiza de mai sus nu este exhaustivă: am utilizat instrucțiuni particulare (de *format R*); desenele simplificate au folosit această cale particulară de date.
- Hazard care poate fi rezolvat cu avansări poate apărea și la *alte combinații* de instrucțiuni: Spre exemplu, la `load` urmat de `store` data de scris în memorie de ia din MEM/WB;
- In MIPS operația `nop` are codul format numai din zerouri, anume este codul pentru `sll $0, $0, $0` (shift left logical cu 0 poziții), instrucțiune care într-adevăr nu modifică nimic;



# Procesorul: Tehnica de pipeline

---

## Cuprins:

- Tehnica de pipeline
- Calea de date cu pipeline
- Controlul pentru implementari cu pipeline
- Hazard de date si avansari
- *Hazard de date si stationari*
- Hazard la ramificatii
- Exceptii
- Concluzii, diverse, etc.



# Hazard de date si stationari

## Intarzieri:

- Există cazuri în care avansarea nu poate rezolva hazardul, e.g., *load*, urmat de *citirea registrului*;
- In astfel de situații trebuie utilizate *întârzieri*, anume procesarea în pipeline este întârziată cu unul ori mai multe cicluri;
- Pentru detectarea acestor hazarduri folosim o *unitate de detectarea hazardului*, care implementează condițiile necesare;
- La exemplu de mai sus (i.e., load urmat de citire registru), hazardul este detectat în faza ID cu condițiile

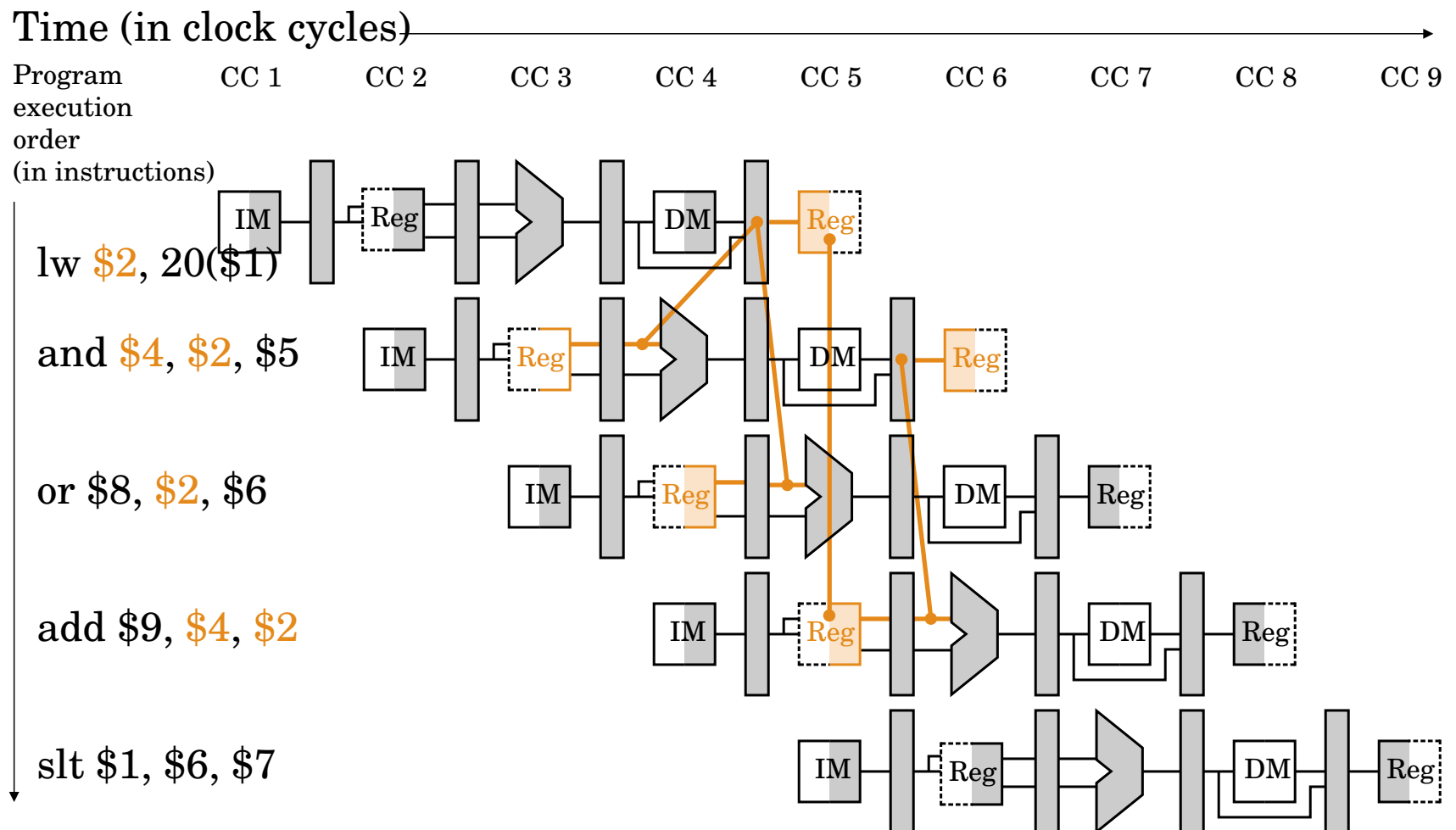
$$\begin{aligned} & [ID/EX.MemRead \quad \wedge \quad (ID/EX.RegisterRt = IF/ID.RegisterRs \\ & \vee \quad ID/EX.RegisterRt = IF/ID.RegisterRt)] \\ & \Rightarrow \quad \text{intarziere pipeline} \end{aligned}$$



# ..Hazard de date si stationari

## Intarzieri (cont.)

- *Exemplu:* o *dependență de date* în pipeline care nu poate fi rezolvată cu avansări, ci cu *întârzieri* ale pipeline-ului:





# Hazard de date si stationari

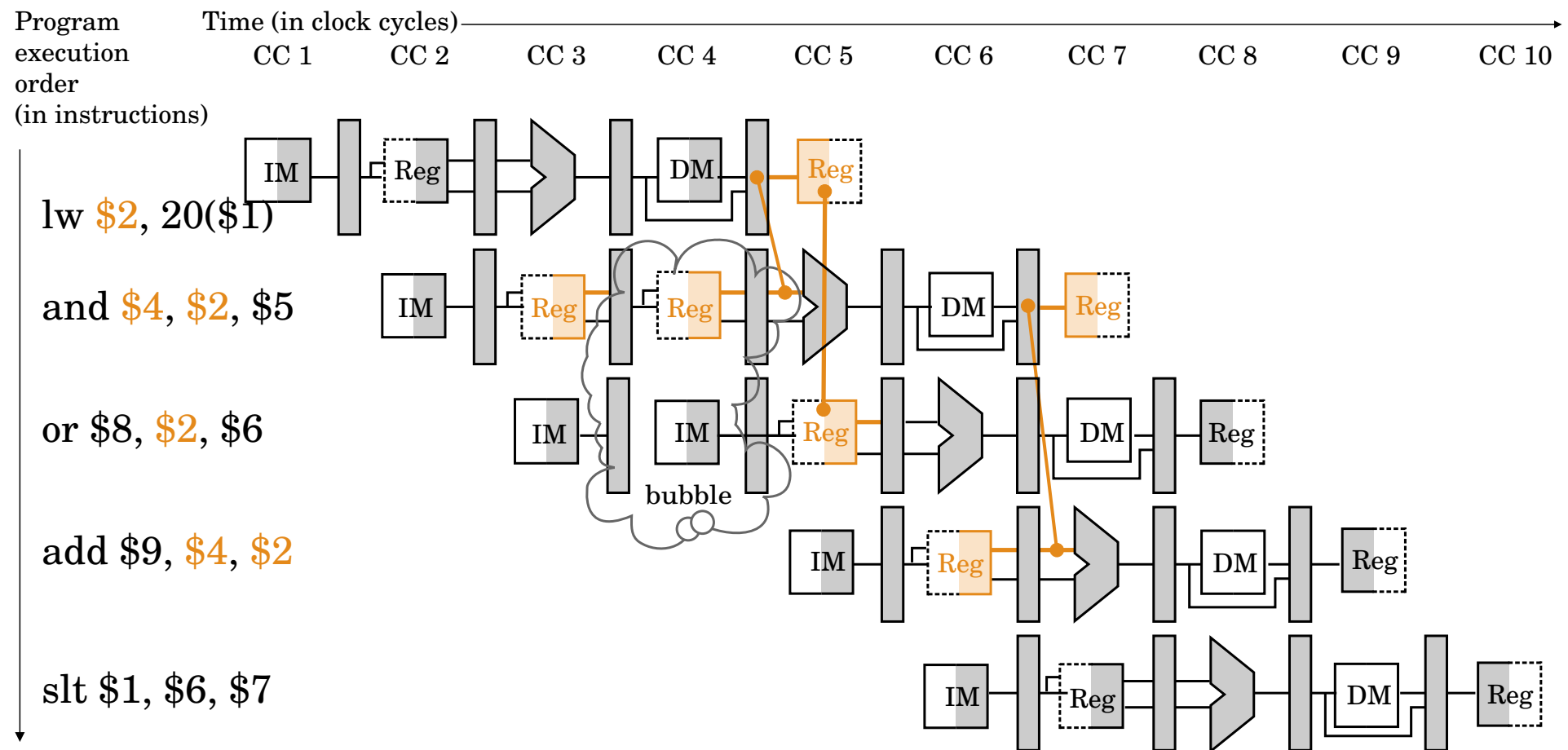
## Intarzieri (cont.)

- Intârzierea trebuie să aibă același efect ca instrucțiunea `nop`, dar acum începând din faza de execuție EX din pipeline;
- Pe exemplul dat întârzierea se simulează astfel:
  - în fazele IF, ID se *refolosesc* același date, *desetând* semnalele `PCWrite`, `IF/IDWrite`;
  - în fazele EX, MEM, WB se *desetează* cele 9 semnalele de control, deci nici regiștrii, nici memoria nu se pot modifica, neputându-se scrie (suficient: `MemWrite`, `RegWrite`);

# ..Hazard de date si stationari

## Intarzieri (cont.)

- Figura ilustrează *rezolvarea cu întârzieri* a dependenței anterioare de date:



## ..Hazard de date si stationari

### Exemplu:

- Ilustrăm cum funcționează tehnica de întârziere pe următoarea secvență de program

lw	\$2, 20(\$1);	(a)
and	\$4, \$2, \$5;	(b)
or	\$8, \$2, \$6;	(c)
add	\$9, \$4, \$2;	(d)
slt	\$1, \$6, \$7;	(e)

- Adăugăm convenția că  $M_a[v]$  (resp.  $M_{+a}[v]$ ) reprezintă valoarea din memorie de la adresa  $v$  la intrarea în (resp. ieșirea din) instrucțiunea  $a$ . Condițiile de corectitudine sunt:

$$M_a[1_a + 20] = 2_{+a} = 2_b = 2_c = 2_d$$

$$2_b \wedge 5_b = 4_{+b} = 4_d$$

## ..Hazard de date si stationari

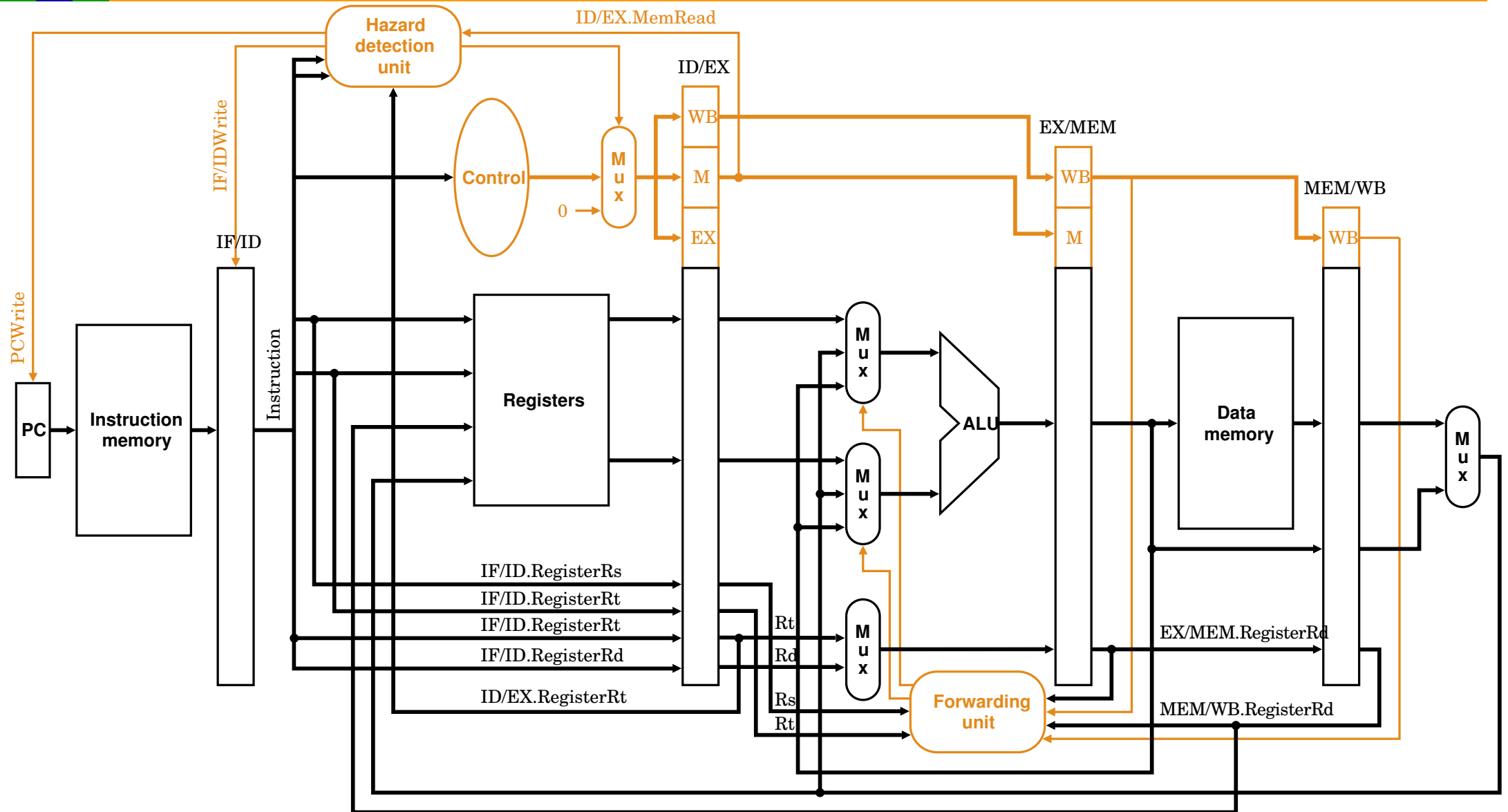
### Exemplu (cont.)

- Tabelul de mai jos indică evoluția calculului:

Ceas	PC	IF/ID	ID/EX(rs,rt)	EX/MEM	MEM/WB
1	$a$				
2	$b$	$I_a$			
3	$c$	$I_b$	$1_a$		
4	$c$	$I_b$	$2_{<b}, 5_b$	$1_a + 20$	
5	$d$	$I_c$	$2_{<b}, 5_b$	$2_{<b} \wedge 5_b$	$M_a[1_a + 20]$
6	$e$	$I_d$	$2_c, 6_c$	$2_b \wedge 5_b$	$2_{<b} \wedge 5_b$
7		$I_e$	$4_{<d}, 2_d$	$2_c \vee 6_c$	$2_b \wedge 5_b$
8			$6_e, 7_e$	$4_d + 2_d$	$2_c \vee 6_c$
9				$6_e \neq 7_e$	$4_d + 2_d$
10					$6_e \neq 7_e$

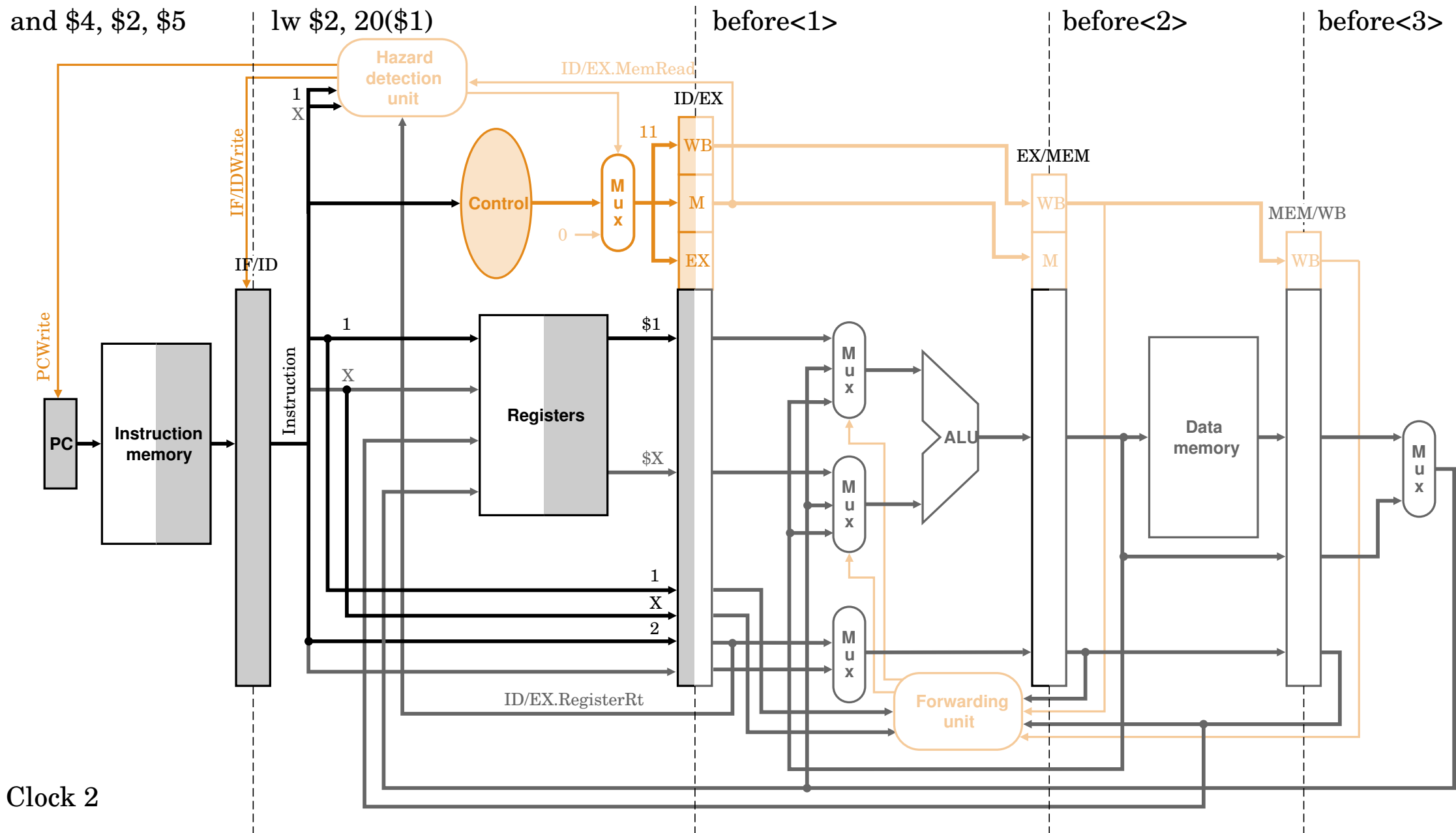
- In pasul 3 detectăm un hazard; procesarea se întârzie cu un ciclu spre a putea folosi valoarea corectă pentru  $2_b$  în pasul 5.

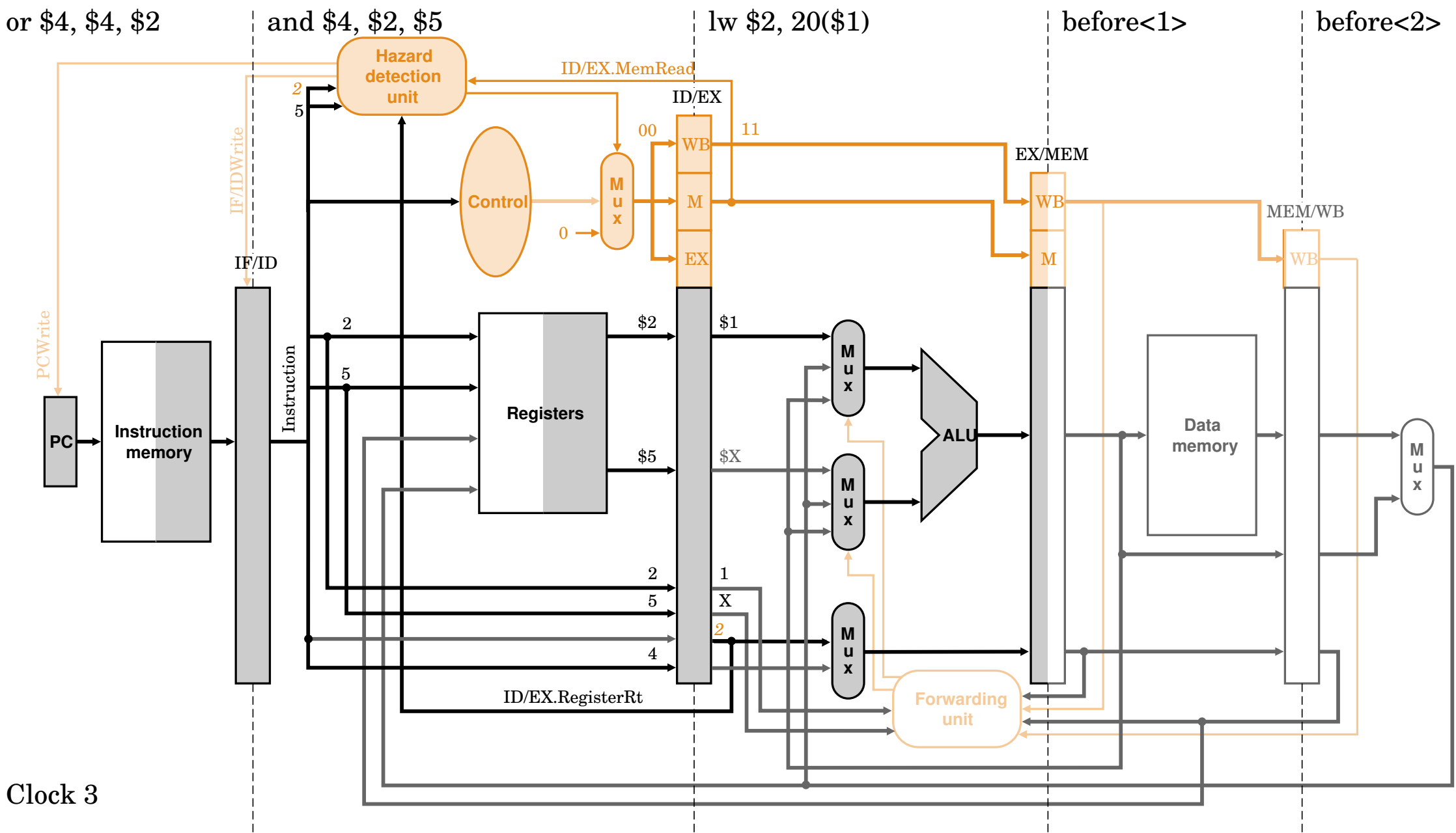
# ..Hazard de date si stationari



Schiță a căii de date cu componenta de *deteție hazarduri*, adăugată la cea pentru avansări (ilustrată pe o cale de date simplificată).

## (Ciclul 2)



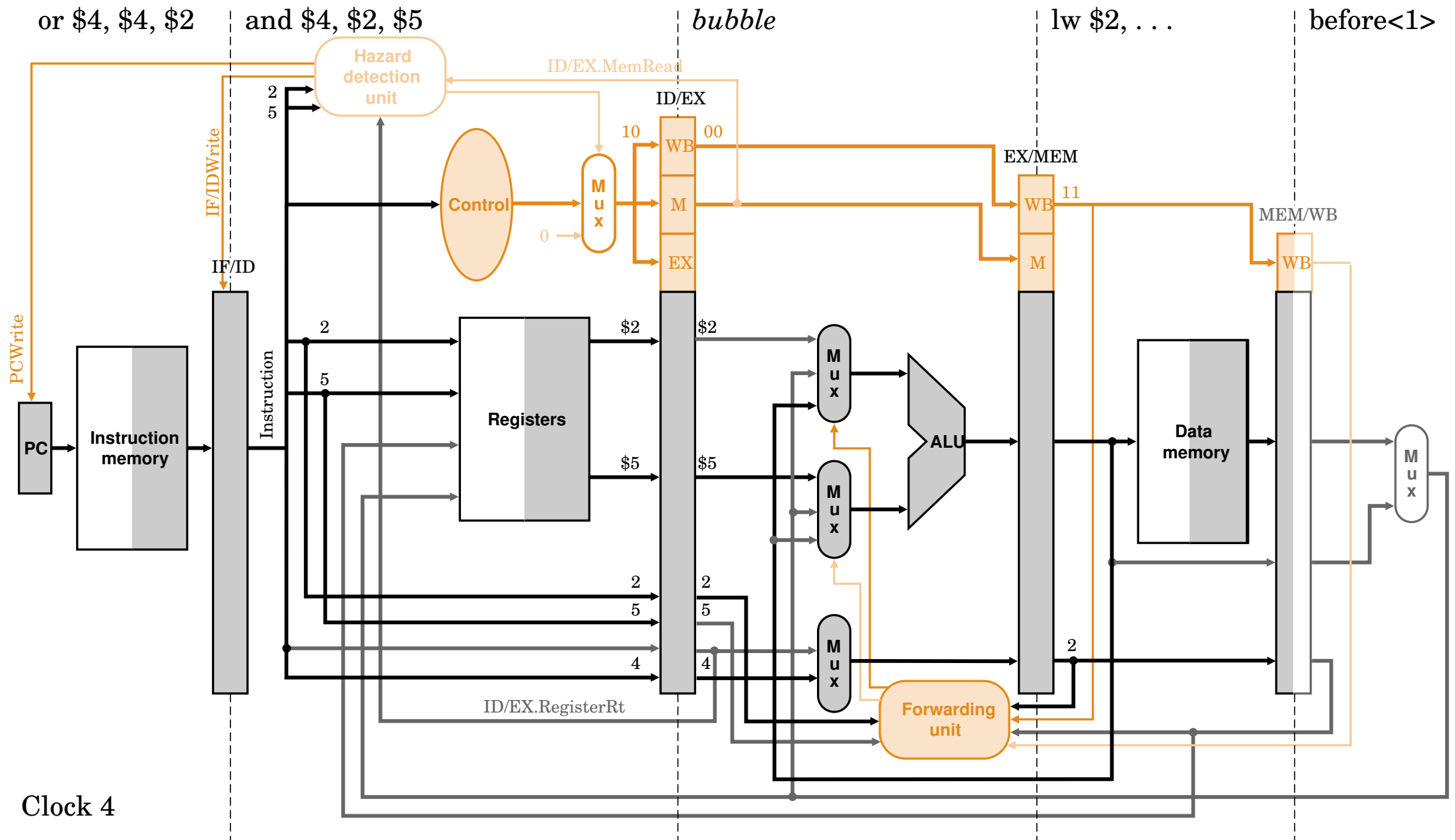


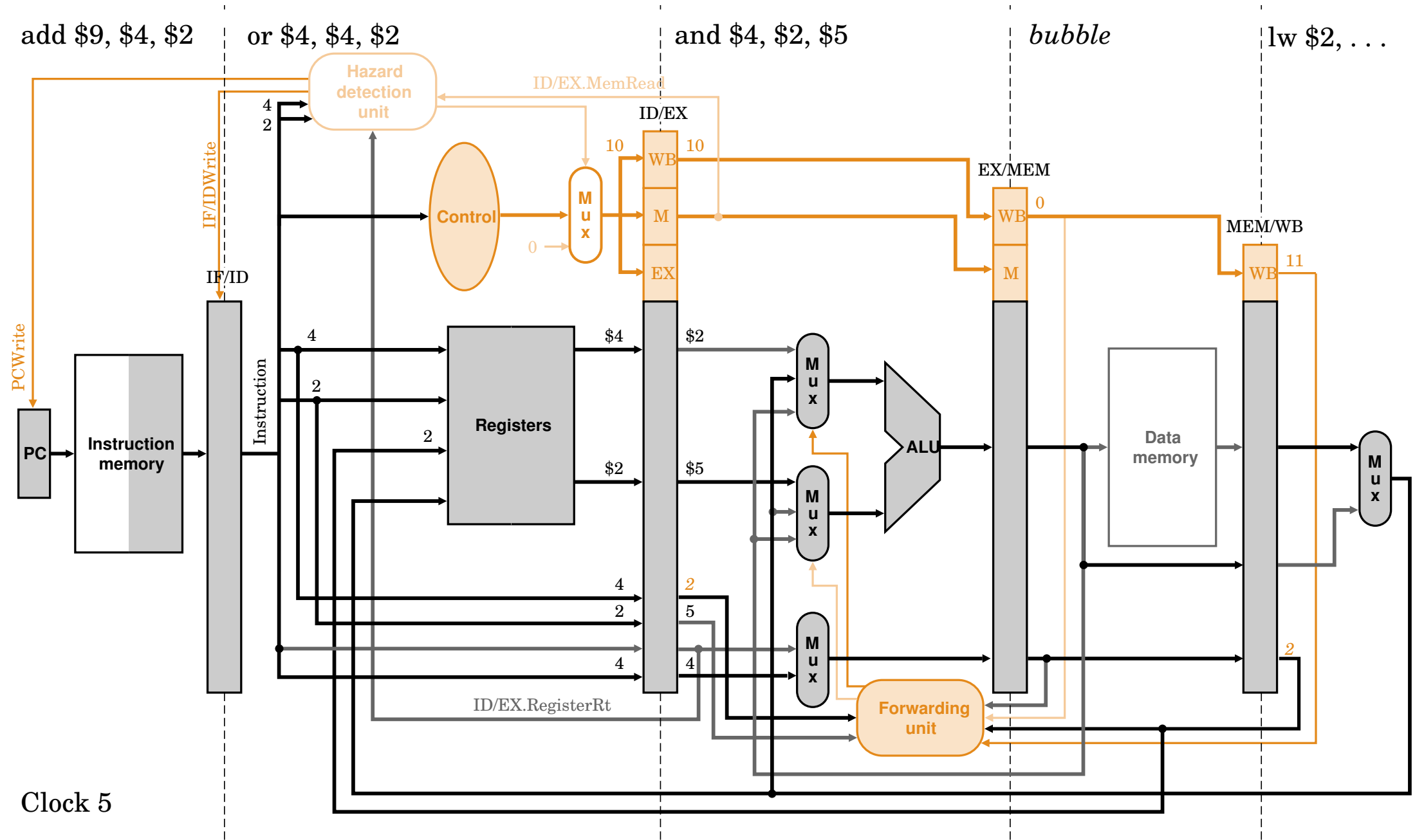
Clock 3

*(Ciclul 3)*



# (Ciclul 4)





*(Ciclul 5)*

***(Ciclul 6)***

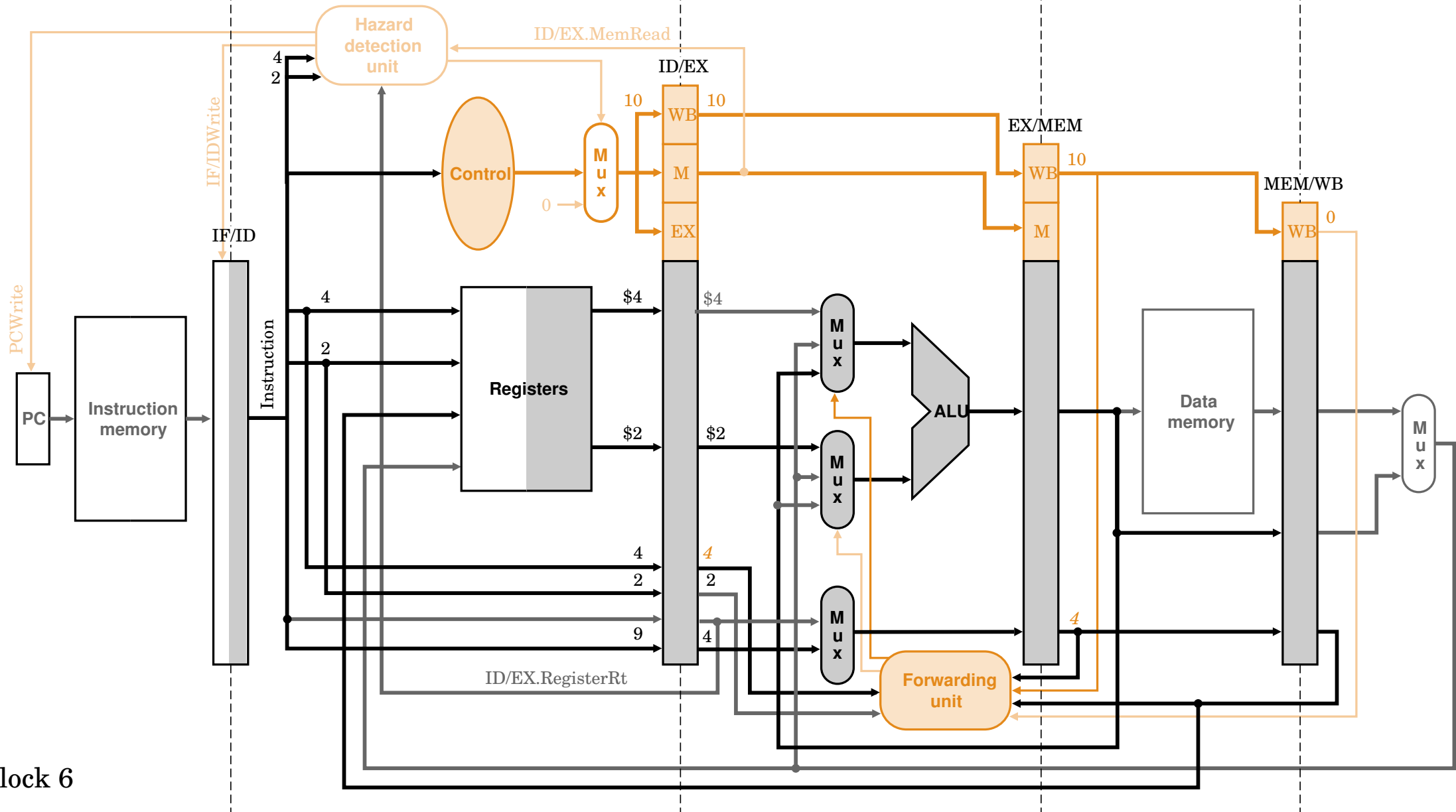
after<1>

add \$9, \$4, \$2

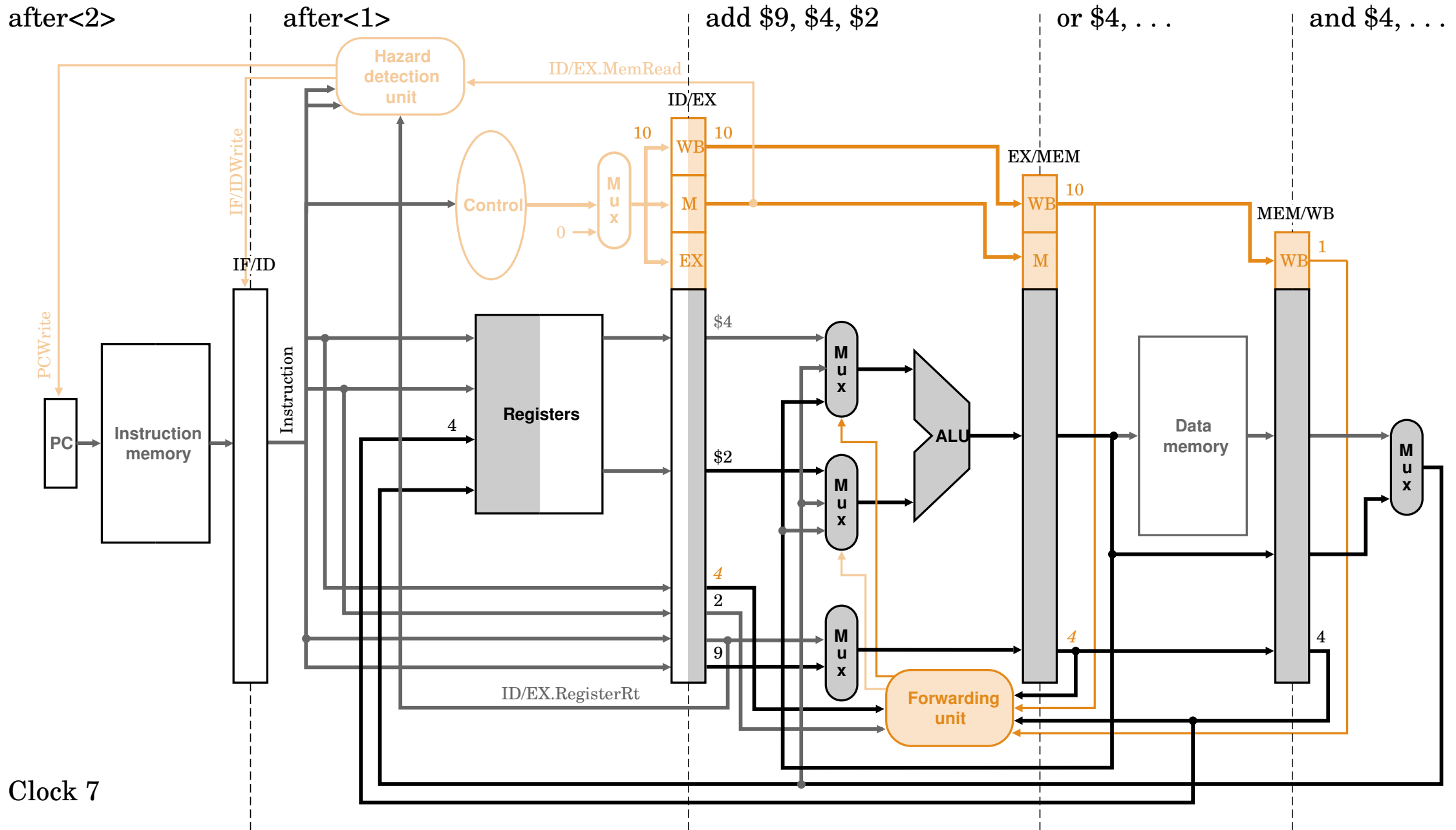
or \$4, \$4, \$2

and \$4, \dots\$

*bubble*



## Clock 6



**(Ciclul 7)**



# Procesorul: Tehnica de pipeline

---

## Cuprins:

- Tehnica de pipeline
- Calea de date cu pipeline
- Controlul pentru implementari cu pipeline
- Hazard de date si avansari
- Hazard de date si stationari
- *Hazard la ramificatii*
- Exceptii
- Concluzii, diverse, etc.



# Hazard la ramificatii

---

## Generalitati:

- In afară de instrucțiunile de format R ori de acces la memorie, *hazarduri* pot produce și instrucțiunile de *ramificare (branch)*.
- Abordarea este mai simplă, căci nu există tehnici generale de detecție și de rezolvare timpurie (precum avansările în cazul hazardul de date).
- In genere, apar mai puțin frecvent, iar tehnicile folosite sunt de a *reduce așteptarea* ori de a *prezice rezultatul*.



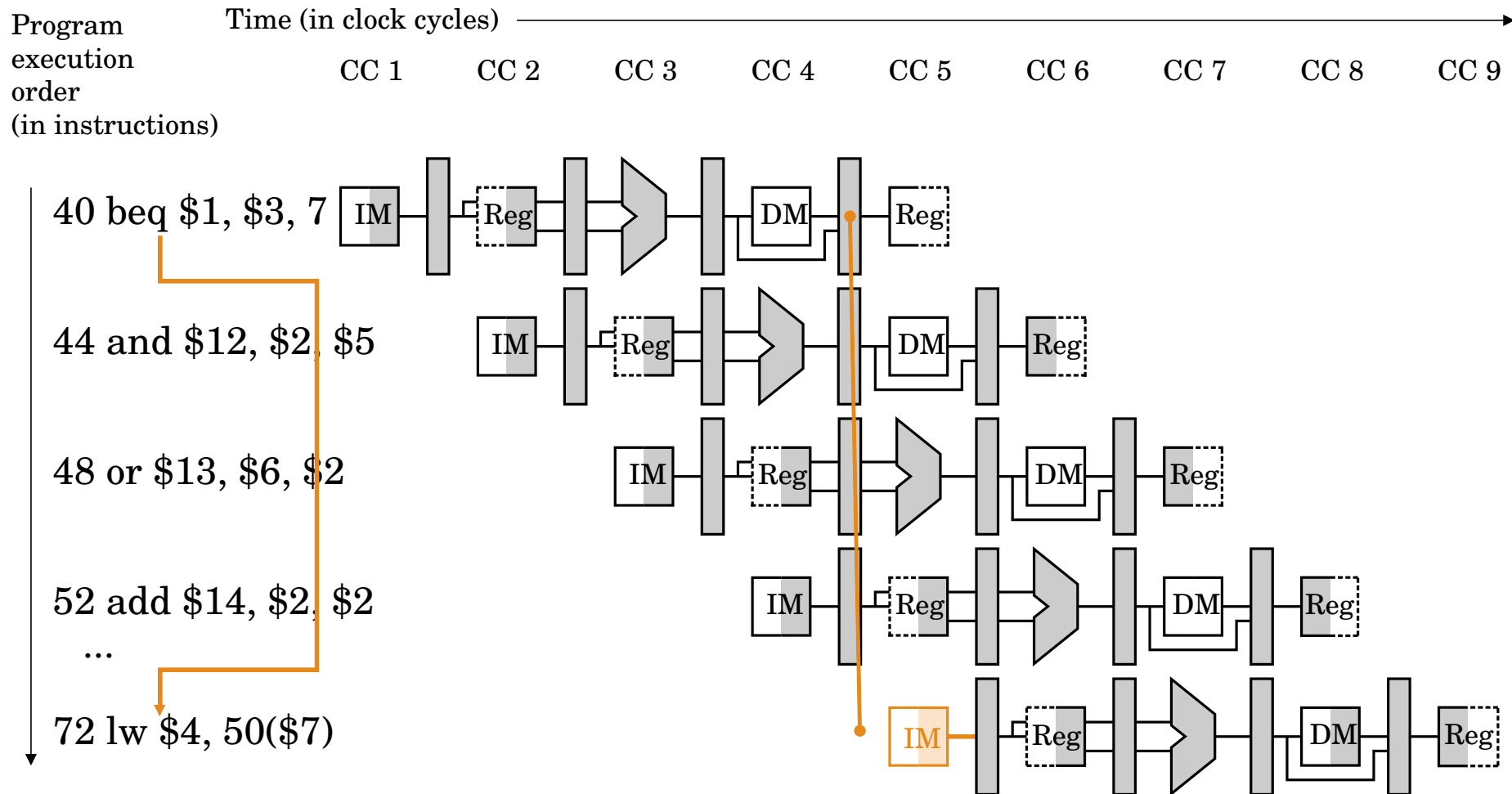
## ..Hazard la ramificatii

---

**Ipoteza de neacceptare a ramificarii:** Procedura bazată pe ipoteza de a *nu accepta ramificarea* este următoarea:

- Se continuă cu secvența următoare.
- Decizia despre test o aflăm abia în faza MEM;
  - dacă decizia este de a accepta testul, se *curăță (flush)* pipeline-ul și se continuă cu instrucțiunea respectivă;
  - dacă nu se acceptă testul se continuă normal.

# ..Hazard la ramificatii



Impactul produs de instrucțiunile *branch* asupra pipeline-ului: se continuă în ipoteza că *ramificația nu este acceptată*; decizia exactă se află abia în faza MEM.



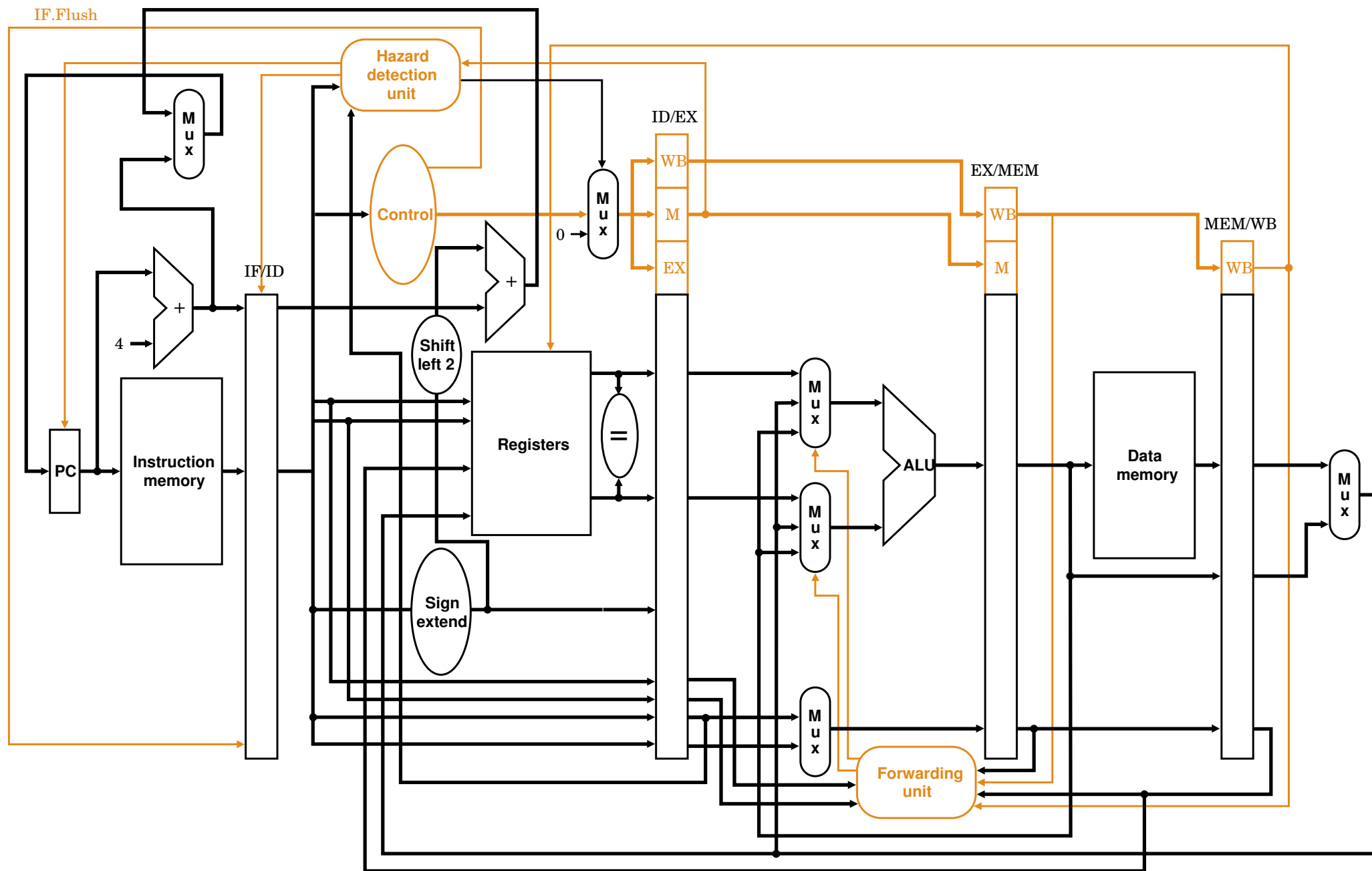


## ..Hazard la ramificatii

### Reducerea întârzierii:

- Uneori, ca în cazul beq, *testul* se poate face *mai simplu* (nu cu scădere ALU): testez egalitatea bit-cu-bit, operație care poate fi făcută în faza ID cu hardware simplu.
- [Complicația este că mutând decizia de test în faza ID și alte componente hardware dependente trebuiesc copiate acolo, spre exemplu cele pentru avansări.]
- Cu cele de mai sus se reduce așteptarea de la 3 cicluri la 1.
- Instrucțiunea care trebuie curățată este acum în faza IF și se curăță ușor *setând la 0 registrul IF/ID* (echivalent cu execuția unei instrucțiuni nop, de cod 0).

# Cale cu branch și hardware pentru flush





# ..Hazard la ramificatii

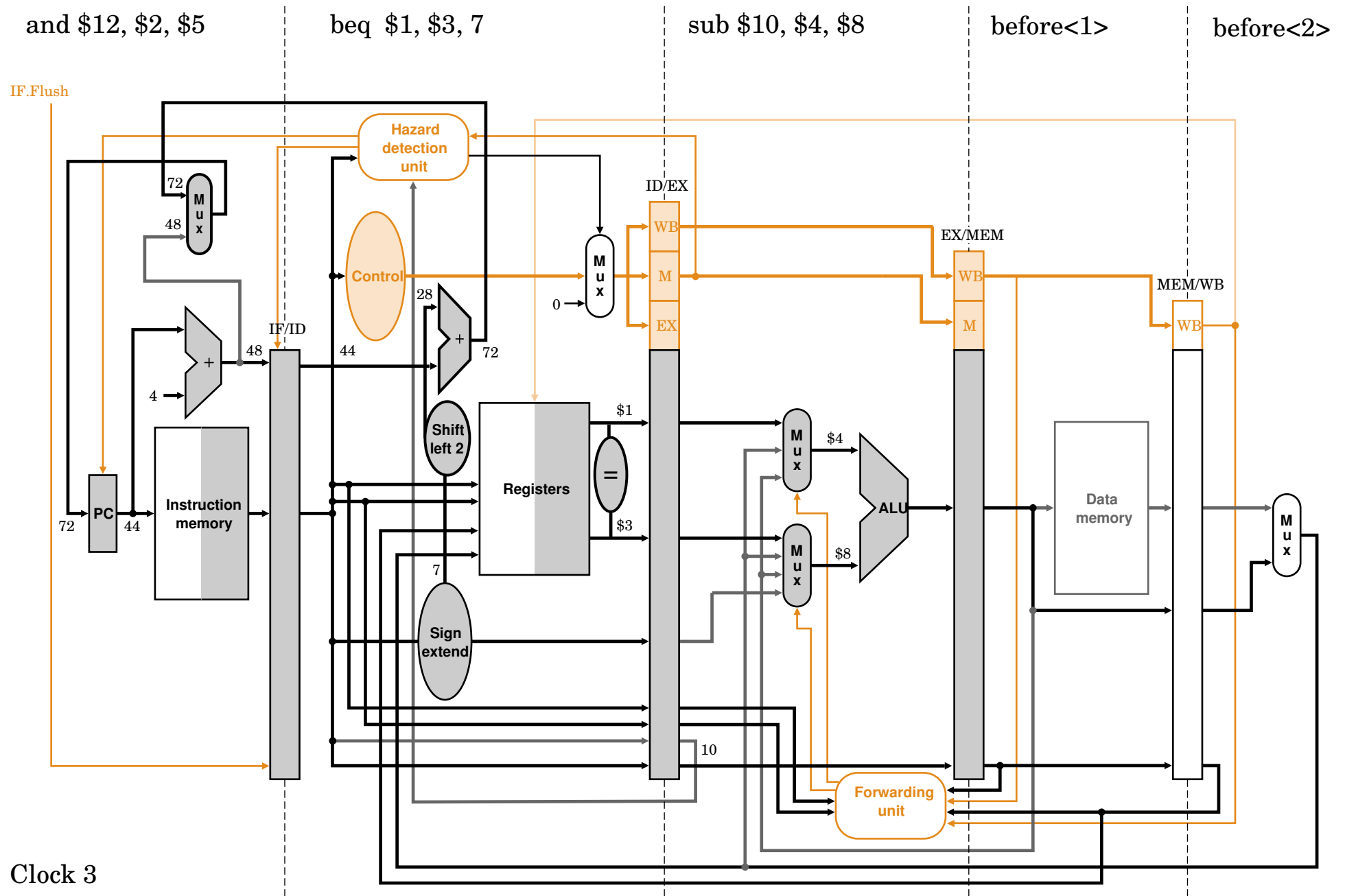
## Exemplu:

- Ilustrăm cum funcționează tehnica pe următoarea secvență de program

```
36 sub    $1, $3, 7;  
40 beq    $10, $4, $8; # salt la 40+4+7*4=72  
44 and    $12, $2, $5;  
48 or     $13, $2, $6;  
52 add    $14, $4, $2;  
...  
72 lw     $4, 50($7);
```

- Desenul indică numai pașii semnificativi (ciclurile 3-4).

# (Ciclul 3)

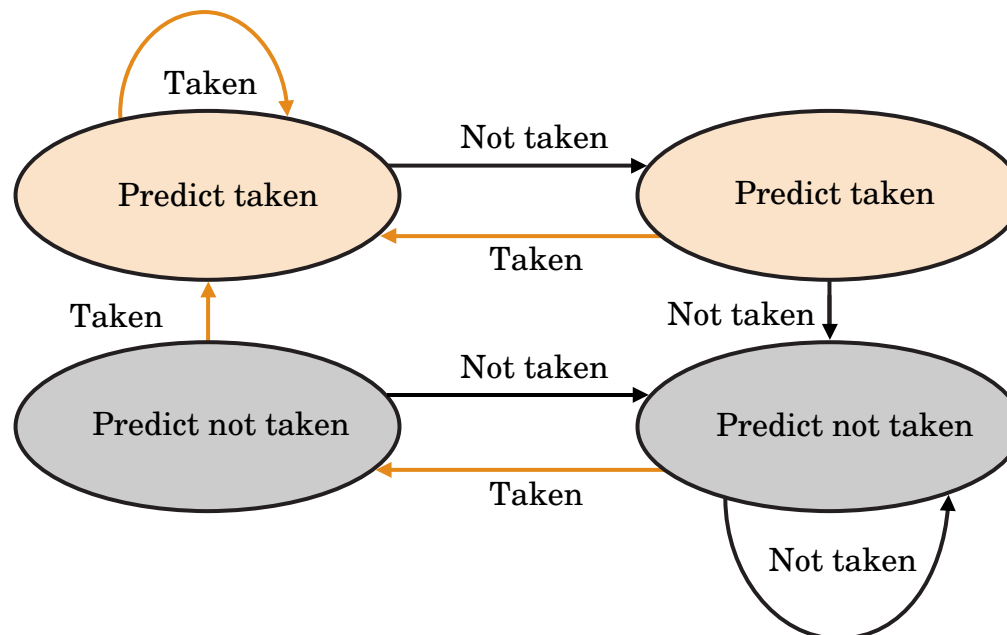




# ..Hazard la ramificatii

## Predictii:

- Continuarea în “ipoteza că nu se acceptă testul” este un caz extrem de *predicție la ramificării*.
- In general, se folosește un *buffer* (tabelă a istoriei selecției) pe baza căruia se face predicția.
- *Exemplu:* O prezicere a selecției la branch care folosește 2 biți (spre a codifica cele 4 stări) - schimbă prezicerea la 2 erori:

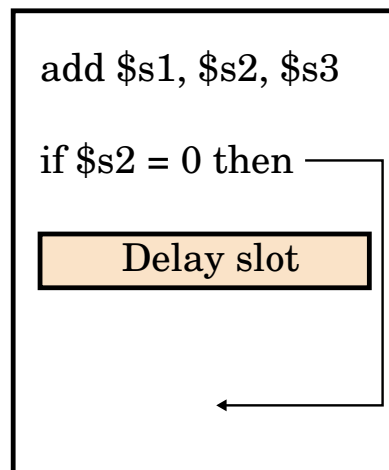


# ..Hazard la ramificatii

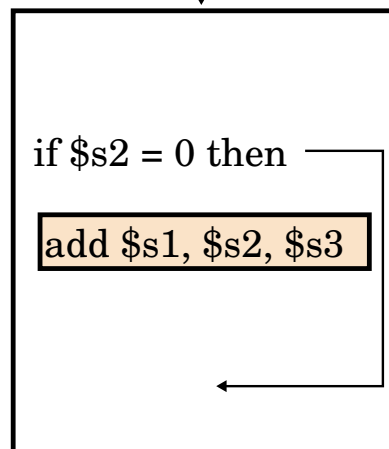
## Slot de intarziere:

Exemple de planificări de activități spre a umple ciclul liber produs de întârzierile generate de ramificații.

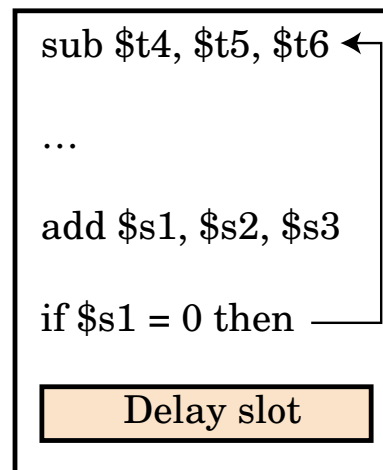
a. From before



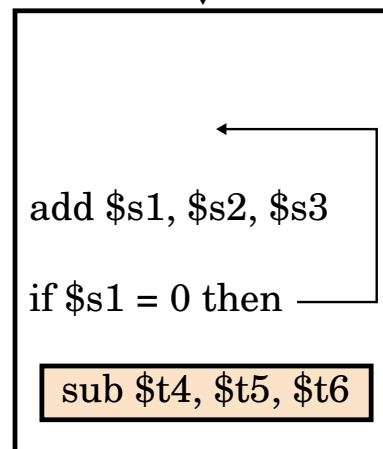
Becomes



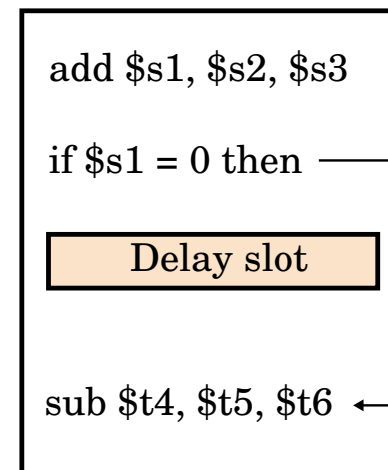
b. From target



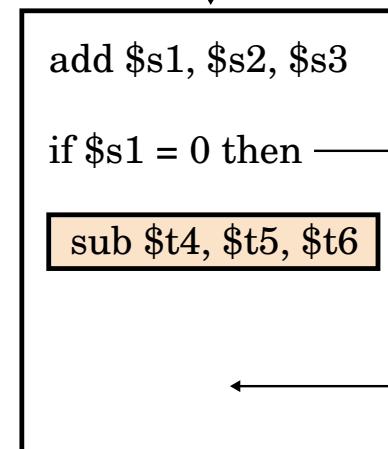
Becomes



c. From fall through



Becomes



(a) este preferabilă; dacă nu se poate, se încearcă (b) ori (c).



# Procesorul: Tehnica de pipeline

---

## Cuprins:

- Tehnica de pipeline
- Calea de date cu pipeline
- Controlul pentru implementari cu pipeline
- Hazard de date si avansari
- Hazard de date si stationari
- Hazard la ramificatii
- *Exceptii*
- Concluzii, diverse, etc.

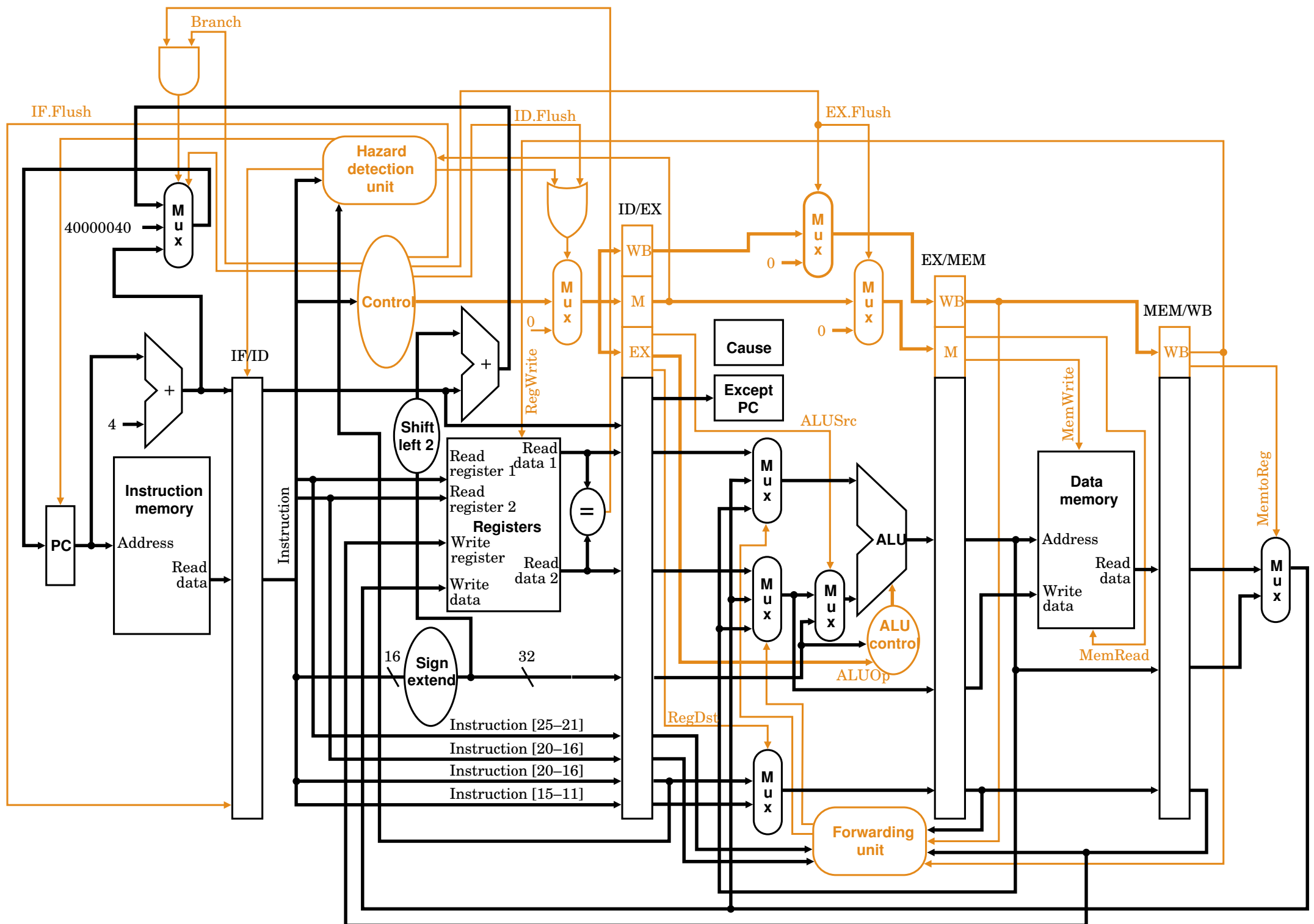


## Exceptii:

- Excepțiile se tratează similar, folosind regiștri EPC (Exception PC) și Cause Register.
- Dacă apare o excepție overflow în ALU, *transferăm controlul* la o adresa  $4000\ 0040_{hex}$  (ce conține adresa procedurii pentru excepții) și *curățăm* pipeline-ul.
- Ultima operație se face cu un semnal de control nou **EX.Flush**, care setează semnalele uzuale de control la zero (suficient: MemWrite și IRWrite, spre a nu modifica memoria ori regiștrii).
- Instrucțiunea care a produs eroarea poate fi mai greu de localizat în pipeline; alternativ, se pot folosi *excepții imprecise* - totuși, MIPS suportă *excepții precise*.

**Cu si fara pipeline:** Pe exemplul anterior (compilatorul gcc) avem:

- *23% load*: 1 ciclu, uzual; 2, dacă este urmat de utilizare; medie: 1.5
- *13% store, 43% ALU*: 1 ciclu
- *19% branch*: 1 ciclu, corect; 2, dacă este predicție greșită; medie: 1.25 (aproximativ 1/4 sunt predicții greșite)
- *2% jump*: 2 cicluri
- *Total*:  $1.5 \times .25 + 1 \times .13 + 1 \times .43 + 1.25 \times .19 + 2 \times .02 = 1.18$
- *Performanță* (ciclu de 2ns):  $8\text{ns (1ciclu)} / 1.18 \times 2\text{ns (pipe)} = 3.40$ .



*Schiță cu implementarea finală de pipeline.*

## Cuprins:

- Tehnica de pipeline
- Calea de date cu pipeline
- Controlul pentru implementari cu pipeline
- Hazard de date si avansari
- Hazard de date si stationari
- Hazard la ramificatii
- Exceptii
- Pipeline superscalar si dinamic
- *Concluzii, diverse, etc.*