Arhitectura sistemelor de calcul

- Prelegerea 12 -

Pipeline

Ruxandra F. Olimid

Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Cuprins

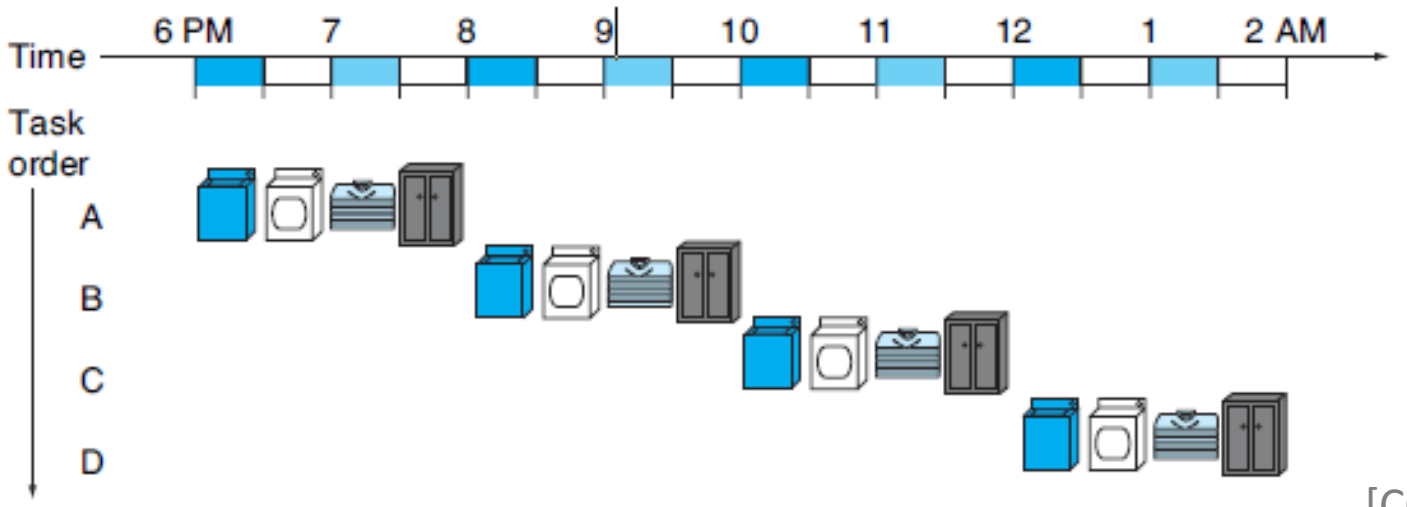
Pipeline

- Tehnica de *pipeline* îmbunătăţeşte eficienţa prin execuţia suprapusă a mai multor instrucţiuni

- Conceptual, tehnica de pipeline:
  - ✓ sparge un proces complex în mai multe etape / faze
  - ✓ dacă fiecare fază necesită resurse diferite, atunci acestea se pot executa concomitent

Pipeline

- Se consideră următorul exemplu, preluat din cartea de curs [COD]:



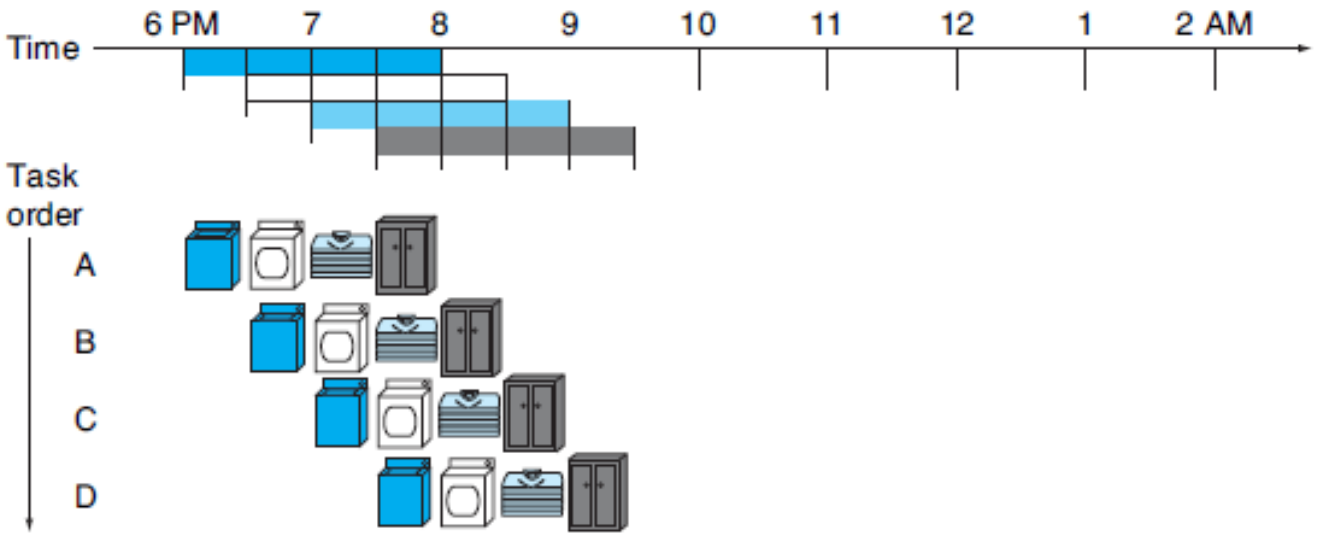
[COD]

- Fiecare etapă (spălare, uscare, călcare, aşezare) se realizează secvenţial, pe fiecare set de rufe

- Când s-a finalizat un set de rufe, se trece la următorul

Pipeline

- Dacă se aplică tehnica de pipeline [COD]:



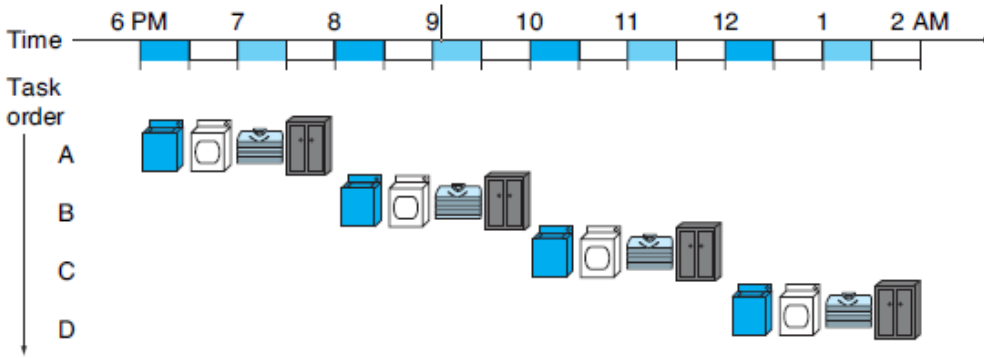
[COD]

- Când pentru un set s-a finalizat o etapă, atunci trece în etapa următoare şi se poate introduce un alt set în etapa iniţială
- Spre exemplu, după ce setul A este spălat trece la uscat, timp în care se poate spăla un alt set B

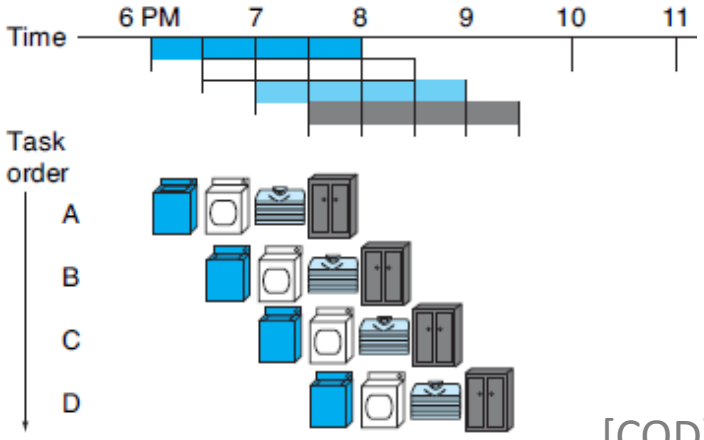
Pipeline

- *Întrebare:* Cât durează spălarea setului A fără se se folosească tehnica de pipeline? Dar când se foloseşte pipeline?

- *Răspuns:* 2h în ambele cazuri



[COD]

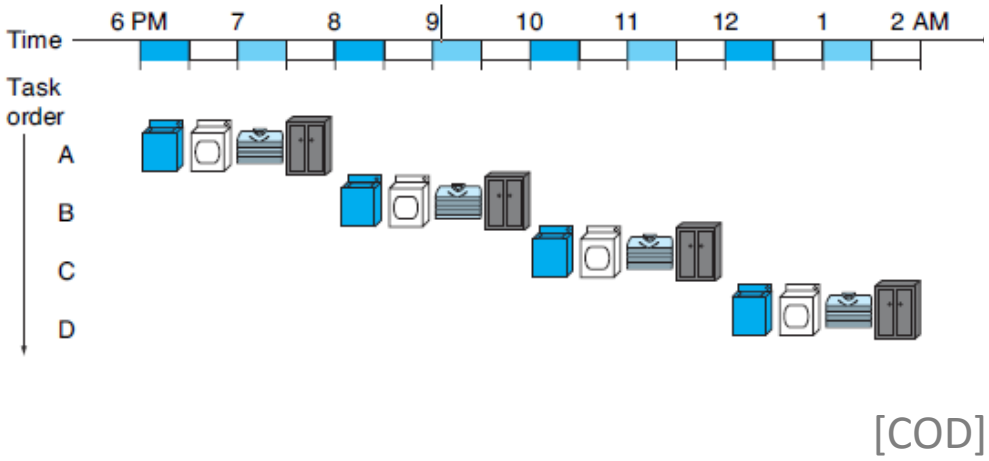


[COD]

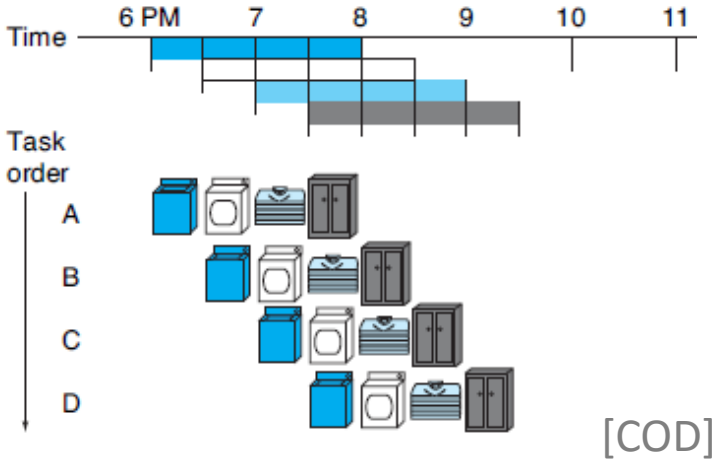
Pipeline

- *Întrebare:* Cât durează spălarea seturilor A, B, C, D fără se se folosească tehnica de pipeline? Dar când se foloseşte pipeline?

- *Răspuns:* 8h, respectiv 3.5h (implementarea pipeline este de aprox. 2.28 mai rapidă)



[COD]

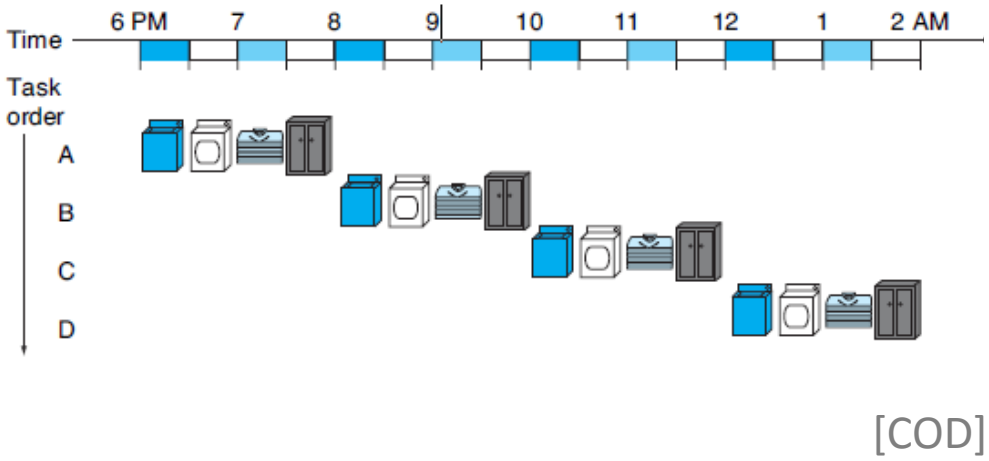


[COD]

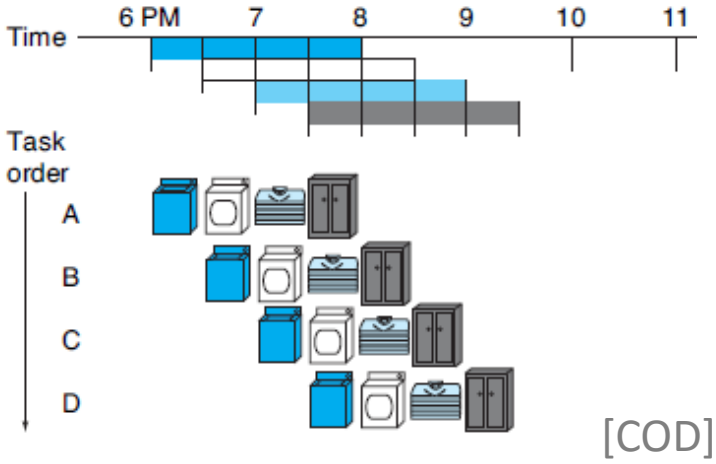
Pipeline

- *Întrebare:* Considerând un proces continuu (un şir infinit de seturi A, B, C...) de câte ori este mai rapidă varianta care implementează pipeline?

- *Răspuns:* De 4 ori (cu excepţia unei întârzieri la început, la fiecare oră se va finaliza un set)



[COD]



[COD]

Pipeline

- Utilizarea pipeline NU ajută dacă volumul procesat este foarte mic (în practică poate chiar întârzia, din cauza unor procesări suplimentare); ex.: timpul necesar pentru finalizarea unui singur set este acelaşi

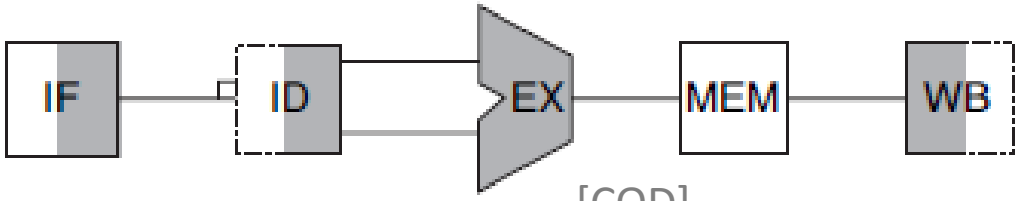
- Pipeline **îmbunătăţeşte productivitatea** (throughput) prin creşterea volumului procesat într-un anumit timp, şi nu prin scăderea timpului necesar pentru o entitate; ex.: timpul pentru un set nu se micşorează, dar în total se termină mai multe seturi de rufe)

Pipeline

- Pentru MIPS tehnica de pipeline prezintă 5 faze:

1. **Încărcarea** instrucţiunilor din memorie – *IF* (*I*nstruction *F*etch)
2. **Decodarea** instrucţiunii şi citirea regiştrilor – *ID* (*I*nstruction *D*ecode)
3. **Execuţia** operaţiei (ex.: `add`) sau calculul unei adrese (ex.: `lw`) – *EX* (*EX*ecution)
4. **Accesarea** memoriei de date (ex.: `sw`) – *MEM* (*MEM*ory access)
5. **Scrierea** rezultatului în registru (ex.: `add`) – *WB* (*W*rite *B*ack)

- **Observaţie!** Nu toate instrucţiunile necesită toţi paşii; ex.: `add` nu foloseşte pasul 4 pentru că nu utilizează memoria de date



[COD]

[Nota: deocamdata facem abstractie de modul de colorare al fazelor instructiunii]

Pipeline

- *Întrebare:* Care paşi sunt necesari pentru fiecare din instrucţiunile `lw`, `sw`, `add`, `beq`?

- *Răspuns:*

	IF	ID	EX	MEM	WB
lw	✓	✓	✓	✓	✓
sw	✓	✓	✓	✓	×
add	✓	✓	✓	×	✓
beq	✓	✓	✓	×	×

Pipeline

- *Întrebare:* Cât durează execuţia fiecărei instrucţiuni din tabelul anterior dacă paşii IF, EX, MEM durează 200ps şi paşii ID şi WB durează 100ps?

- *Răspuns:*

	IF	ID	EX	MEM	WB	Total
lw	200	100	200	200	100	800
sw	200	100	200	200	×	700
add	200	100	200	×	100	600
beq	200	100	200	×	×	500

Pipeline

- *Întrebare:* Folosind valorile de la exerciţiul anterior, cât durează execuţia programului următor dacă se foloseşte implementarea cu un singur ciclu de lungime fixă (i.e. orice instrucţiune se execută într-un singur ciclu de ceas)?

```
lw $s1,0($s2)
lw $s3, 0($s4)
add $s5, $s1, $s3
```

- *Răspuns:* Lungimea unui ciclu este determinată de timpul maxim de execuţie a celei mai lente instrucţiuni, deci este 800ps. Programul are 3 instrucţiuni, deci necesită 3 x 800 = 2400ps

Pipeline

- *Întrebare:* Dar folosind tehnica pipeline?

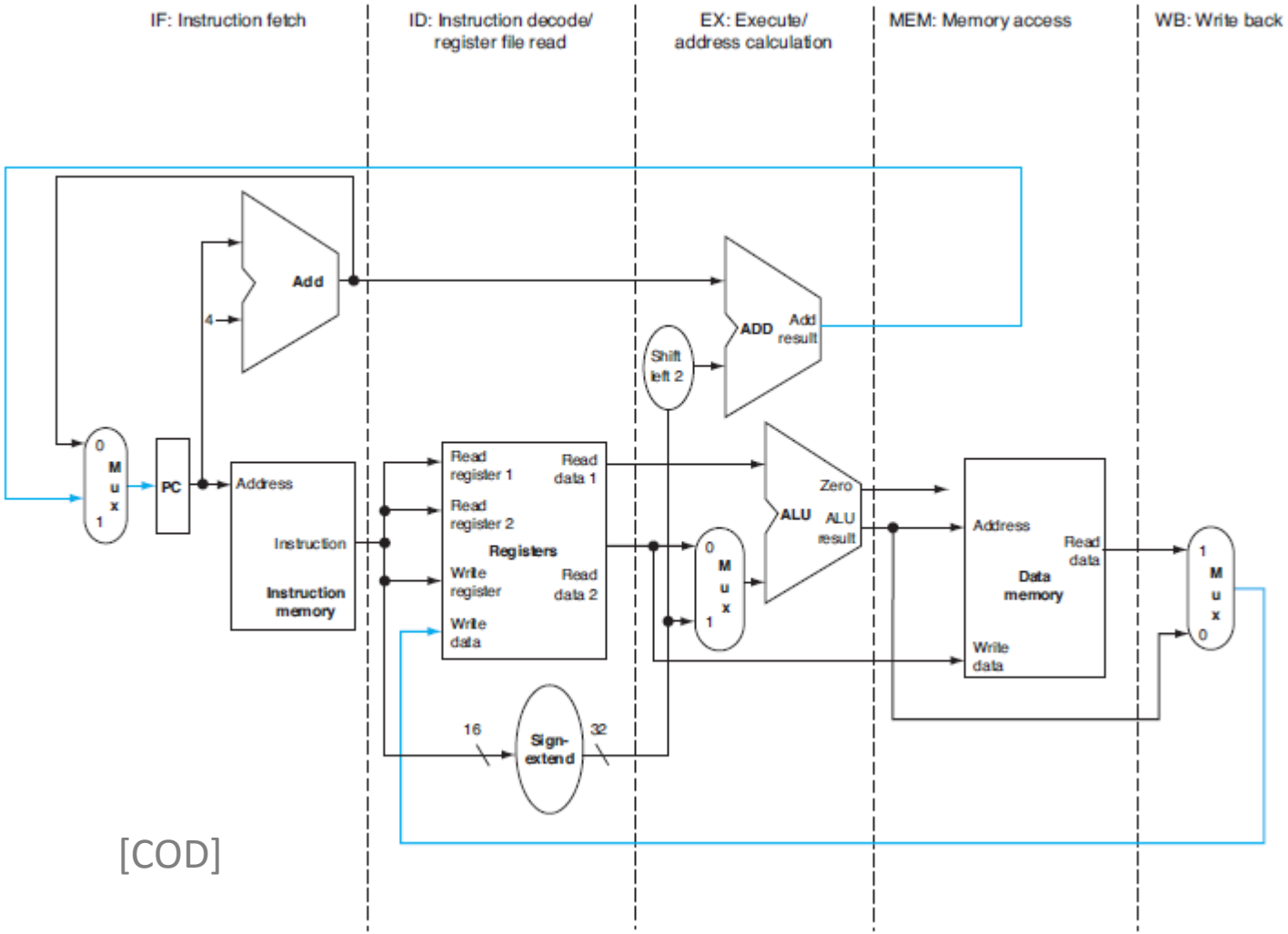
- *Răspuns:* Lungimea unui ciclu este determinată de timpul maxim de execuţie al unui pas, deci este 200ps. Programul are 3 instrucţiuni, conform pipeline necesită 5 ciclii pentru finalizarea execuţiei primei instrucţiuni, apoi cate 1 ciclu pentru execuţia fiecărei intrucţiuni următoare. În final: 7 x 200 = 1400ps

Pipeline

- Definirea setului de instrucţiuni MIPS s-a realizat pentru a utiliza tehnica de pipeline:
  - ✓ Instrucţiunile au dimensiune fixă, ceea ce simplifică încărcarea şi decodarea lor în paşii 1 şi 2
  - ✓ MIPS prezintă doar 3 formate de instrucţiuni (R, I, J) şi păstrează aceeaşi locaţie pentru registrul sursă (R, I), ceea ce permite determinarea formatului şi citirea registrului sursă simultan
  - ✓ Lucrul cu memoria se foloseşte doar pentru instrucţiunile de tip load si store, când se calculează adresa, apoi în etapa următoare se accesează memoria; dacă s-ar fi permis operarea cu valori direct din memorie erau necesari paşi suplimentari
  - ✓ Operanzii sunt aliniaţi în memorie, deci pentru `lw` şi `sw` nu este necesar decât un singur acces la memorie

Implementare (II)

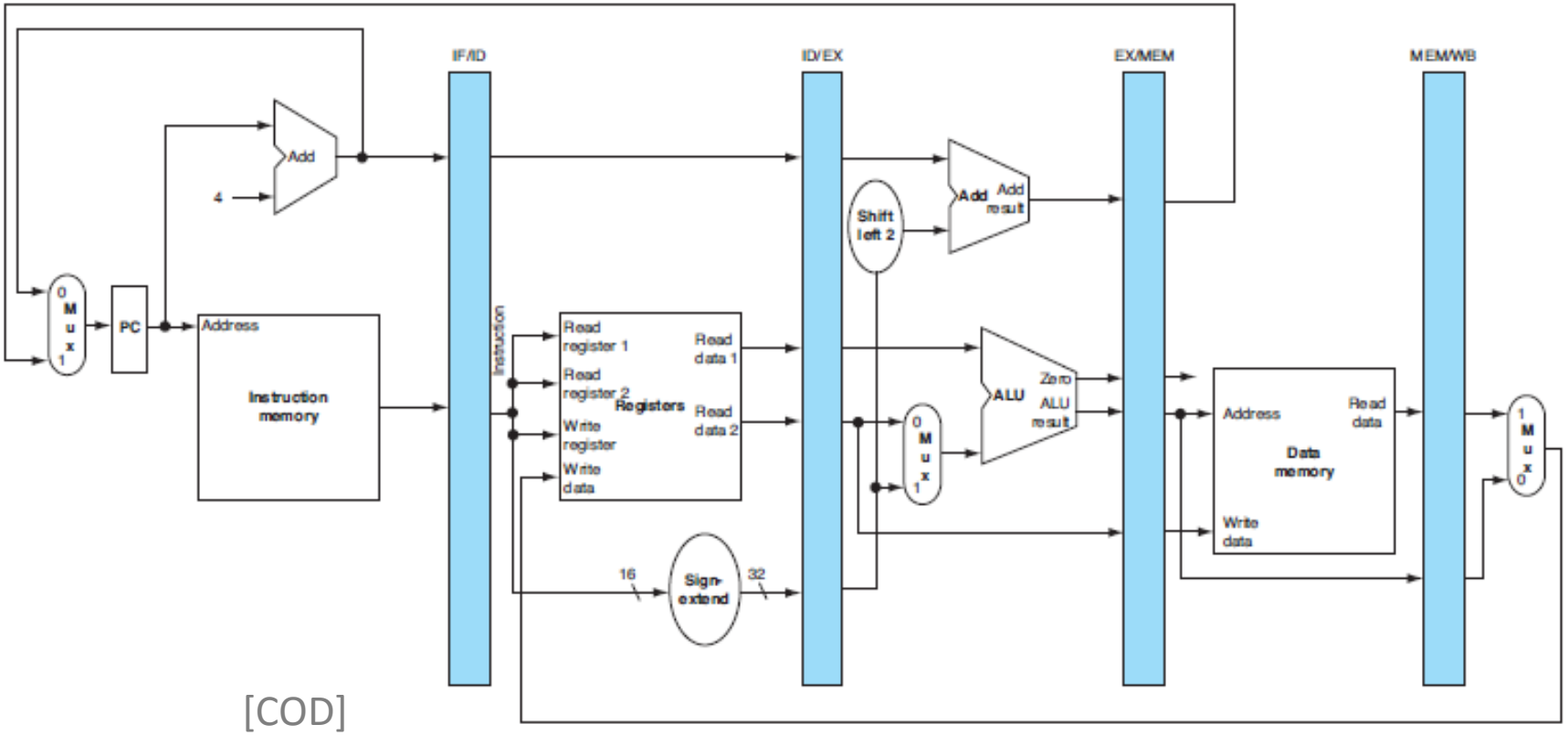
- Evidenţierea fazelor de pipeline:



[COD]

Implementare (II)

- Se introduc regiştrii necesari să reţină datele între fazele de pipeline:



[COD]

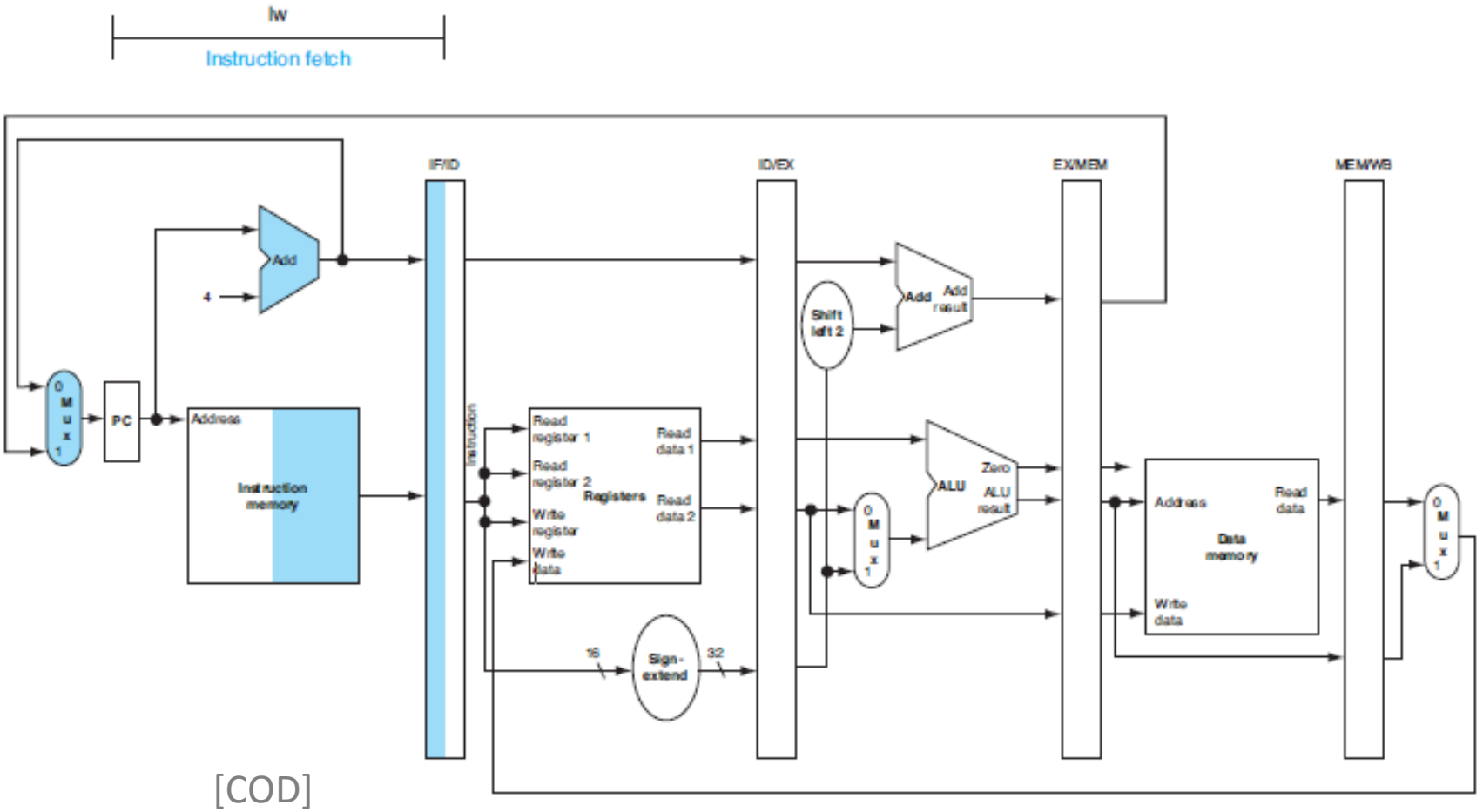
Implementare (II)

- Se reprezintă colorat:
  - ✓ partea stângă a regiştrilor la scriere
  - ✓ partea dreaptă a regiştrilor la citire

- Trecem prin toate fazele instrucţiunea `lw`

Implementare (II)

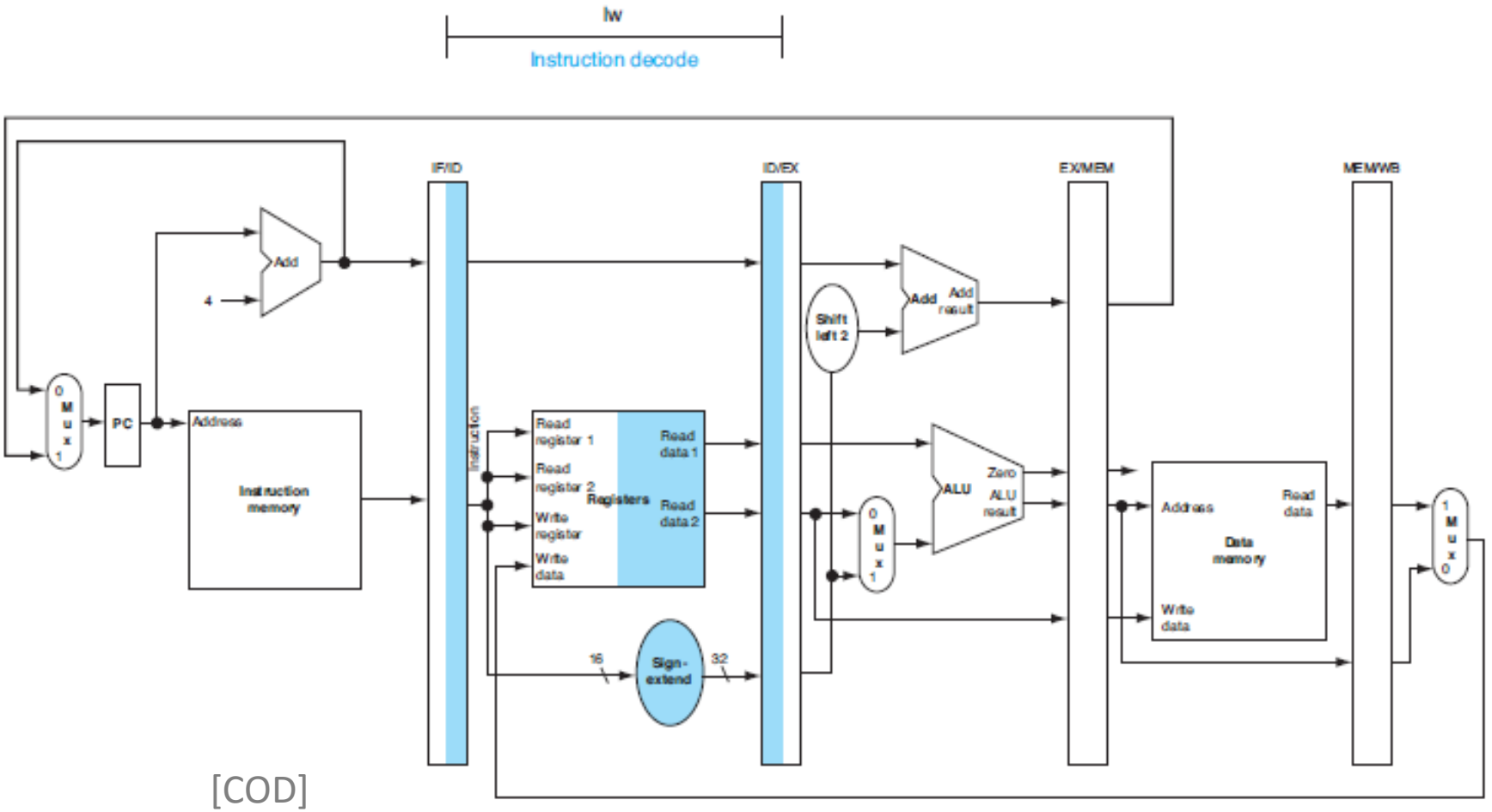
- IF (Instruction Fetch):



[COD]

Implementare (II)

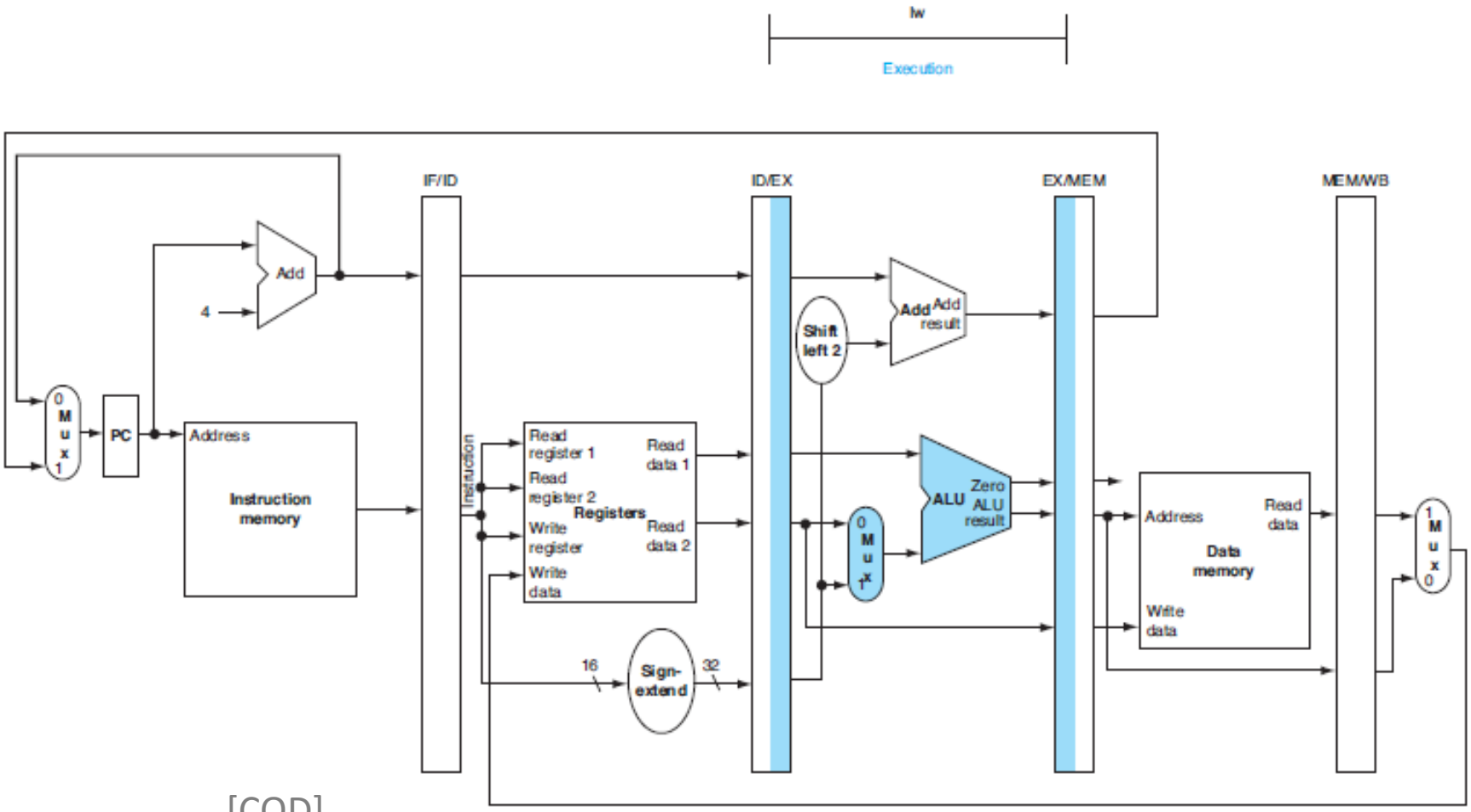
- ID (Instruction Decode):



[COD]

Implementare (II)

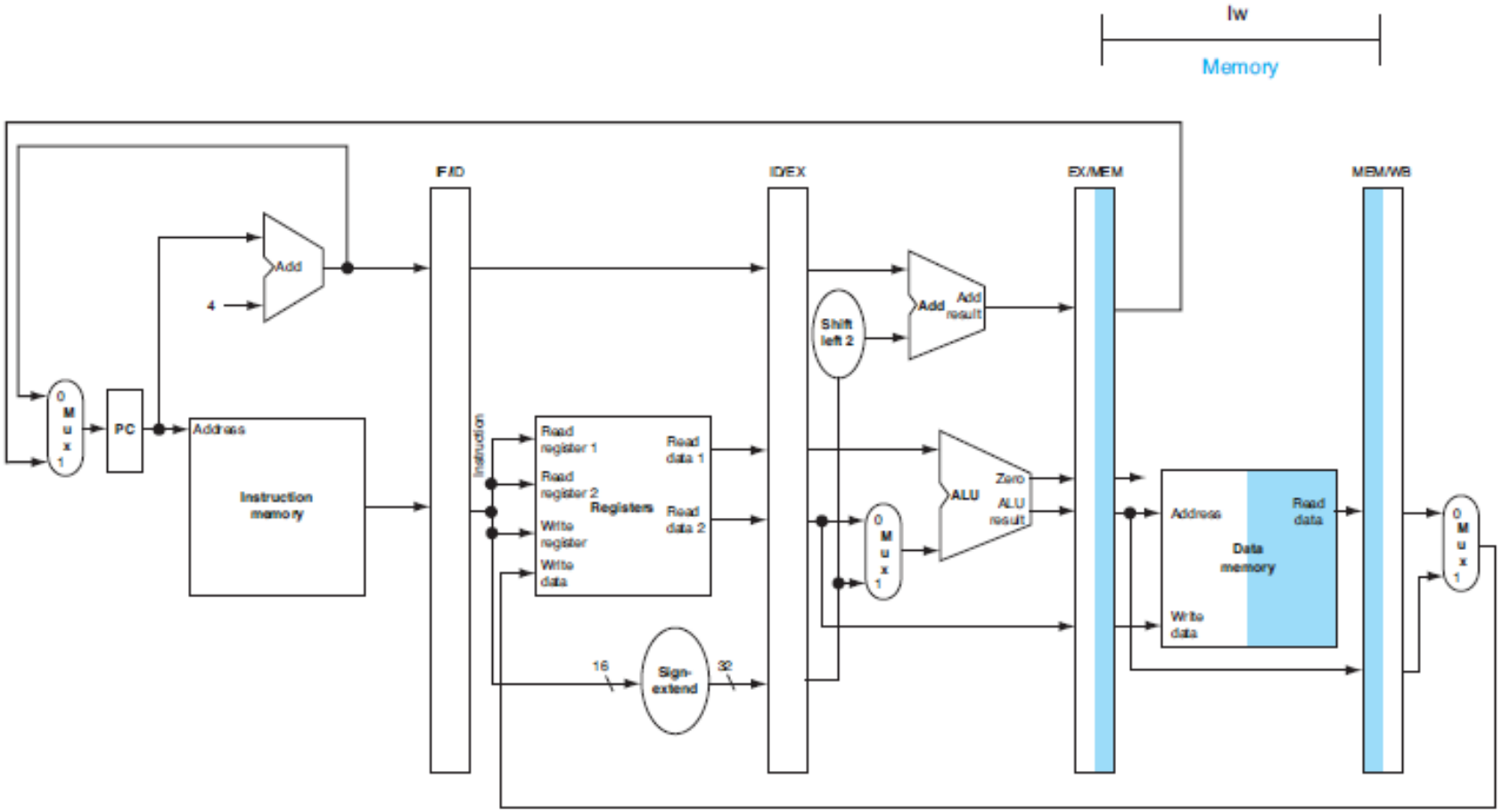
- EX (EXecute):



[COD]

Implementare (II)

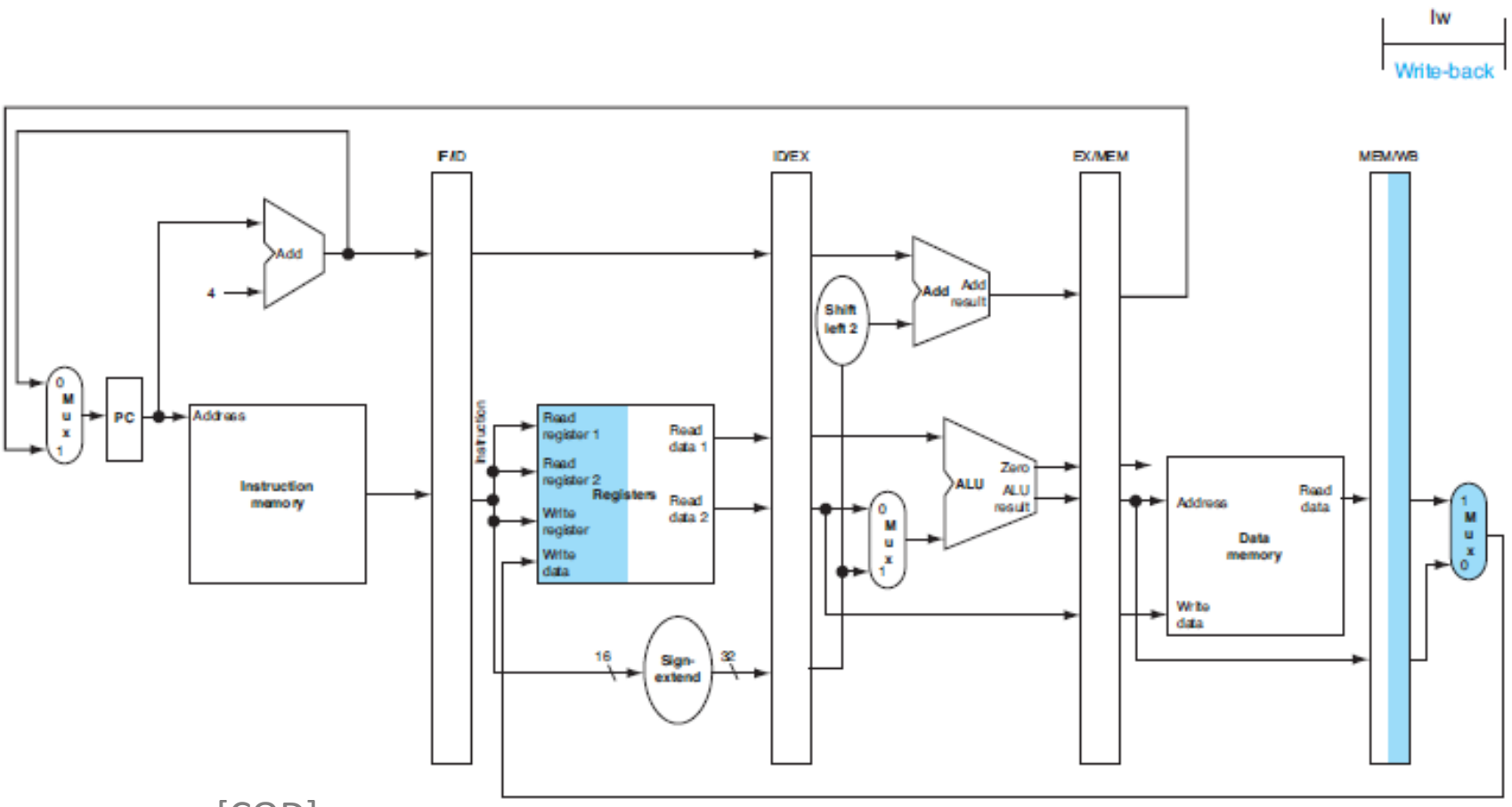
- MEM (MEMory access):



[COD]

Implementare (II)

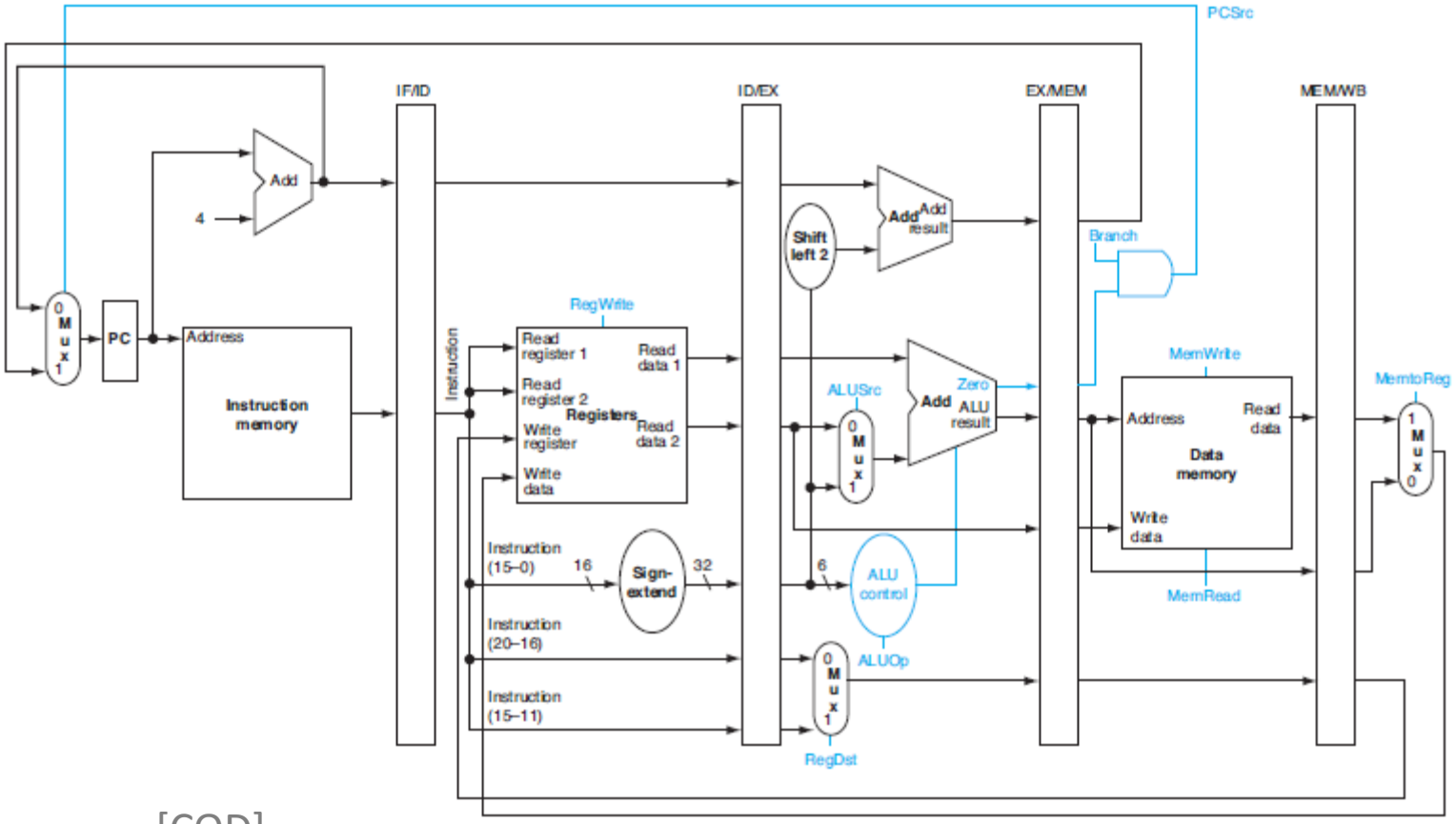
- WB (Write Back):



[COD]

Unitatea de control

- Pornind de la schema generală, evidenţiem semnalele de control:



[COD]

Unitatea de control

- Semnalele de control sunt cele de la prima implementare a procesorului

- Se consideră că se realizează la fiecare ciclu, deci nu necesită semnale de control suplimentare:
  - ✓ citirea din memorie
  - ✓ scrierea în PC şi regiştrii de pipeline IF/ID, ID/EX, EX/MEM, MEM/WB

- Analizând schema anterioară, se diferenţiază imediat semnalele de control specifice fiecărei etape pipeline:
  - ✓ **IF , ID**: nu necesită semnale de control (citirea din memorie şi scrierea în regiştrii se face la fiecare tact)
  - ✓ **EX**: *RegDst*, *ALUOp* (2 biţi), *ALUSrc*
  - ✓ **MEM:** *Branch*, *MemRead*, *MemWrite*
  - ✓ **WB:** *MemtoReg*, *RegWrite*

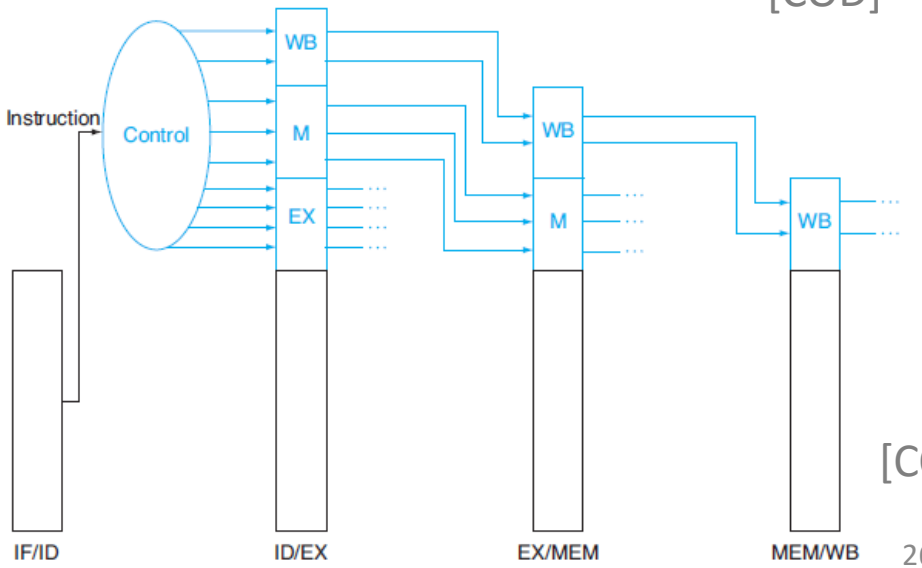
Unitatea de control

- Semnalele unităţii centrale de control se pot grupa în funcţie de etapele pipeline:

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

[COD]

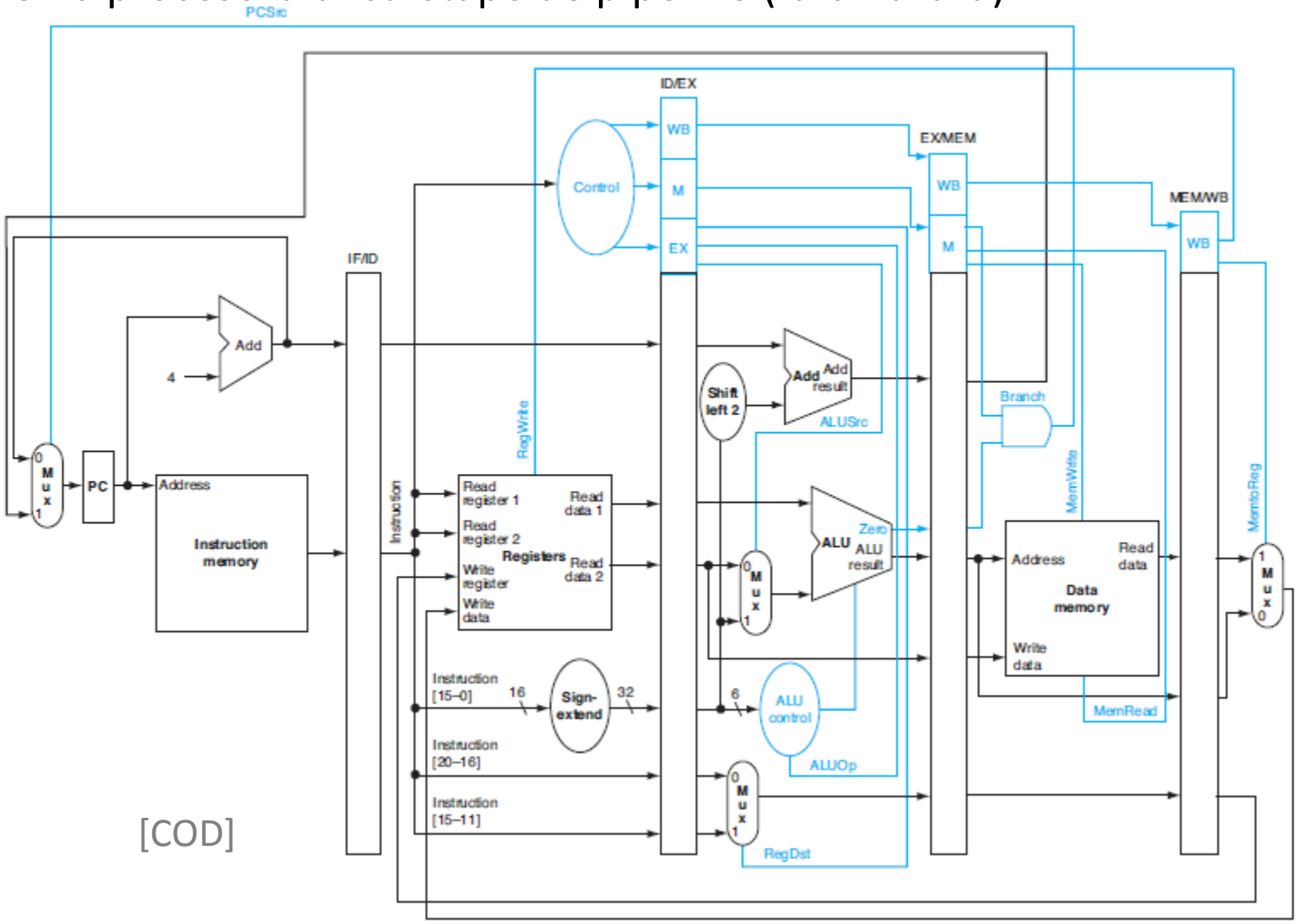
- Acestea se memorează în regiştrii de pipeline şi se transmit dintr-o fază în următoarea:



[COD]

Implementare (II)

- Schema procesorului cu etape de pipeline (fără hazard)



[COD]

Hazard

- Există situaţii când o etapă de procesare a unei instrucţiuni nu se poate executa în următoarea etapă din pipeline

- O astfel de situaţie poartă denumirea de *hazard*

- Exista 3 tipuri de hazard:
  - ✓ *Hazard structural* (**restricţie fizică**) : cauzat de hardware

  ex.: se foloseşte aceeaşi componentă hardware pentru 2 etape succesive
  - ✓ *Hazard de date* (**restricţie logică**) : cauzat de dependenţa unei variabile (registru, valoare, etc.)

  ex.: nu se cunoaşte încă valoarea unui registru, dar se foloseşte într-o altă instrucţiune
  - ✓ *Hazard de control* (**restricţie logică**): cauzat de instrucţiunile de salt

  ex.: trebuie luată o decizie pe baza unui rezultat incă necalculat, se încarcă (IF) următoarea instrucţiune, dar nu aceasta este cea care trebuie executată următoarea

Hazard structural

- *Hazardul structural* apare când nu se pot executa toate operaţiile necesare într-o etapa de pipeline din cauza unor restricţii hardware

- Fiindcă este introdus de o limitare hardware, este o restricţie **fizică**

- În exemplul iniţial (spălatul rufelor), apare hazard structural dacă:
  - ✓ aceeaşi maşină realizează spălarea şi călcarea
  - ✓ aceeaşi persoană calcă şi aşează rufele

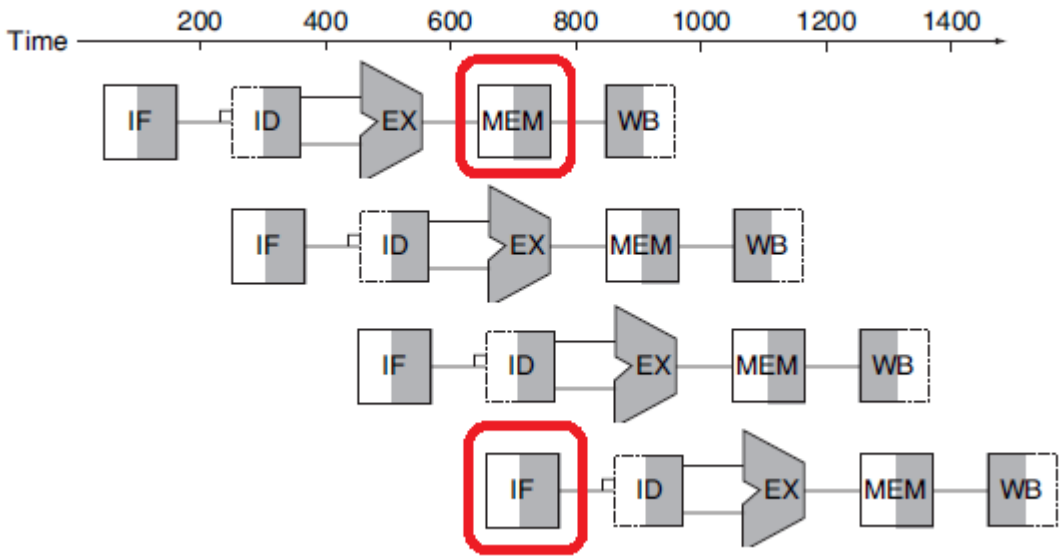
- Pentru un procesor cu o memorie comună de date şi instructiuni (care să nu poată realiza operaţii paralele de acces), un exemplu de hazard structural este când se încearcă scrierea simultană în memorie (sau citirea simultană din memorie)

Hazard structural

- *Întrebare:* Unde apare hazardul structural ?

```
lw $s1,0($s1)
lw $s2, 0($s2)
lw $s3,0($s3)
lw $s4, 0($4)
```

- *Răspuns:* La al 4-lea tact, pentru instr.1 citeşte datele din memorie şi instr.4 se încarcă în memorie (avem deci acces simultan pentru citire din memorie)



Hazard de date

- *Hazardul de date* apare când o instrucţiune nu se poate executa în tactul de ceas corespunzător pentru că datele necesare execuţiei nu sunt încă disponibile

- Fiindcă nu este introdus de o limitare hardware, este o restricţie **logică**

- În exemplul iniţial (spălatul rufelor), apare hazard de date dacă:
  - ✓ la etapa de aşezare a rufelor se găseşte o şosetă fără pereche (trebuie să se aştepte perechea pentru a putea fi strânse şi aşezate corespunzător la loc)

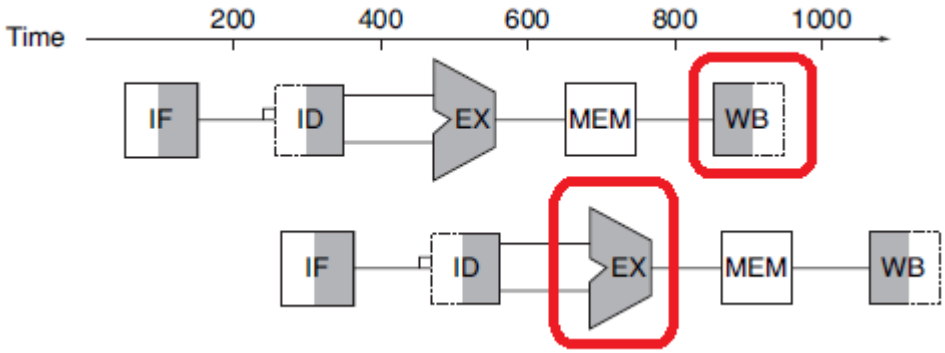
- În calculator, hazardul de date apare dacă se foloseşte o valoare care încă nu este calculată

Hazard de date

- *Întrebare:* Unde apare hazardul de date ?

```
add $t0, $t0, $t1
add $t4, $t0, $t3
```

- *Răspuns:* Valorea sumei pentru instr.1 se scrie în registrul `$t0` în etapa WB, în timp ce instr.2 necesită valoarea în etapa EX, care este anterioară ca timp



Hazard de date

- O metoda de rezolvare a hazardului de date este *forwarding* (sau *bypassing*; sau *tehnica de avansare*): valorile se preiau din regiştrii care delimitează cele 5 etape pipeline (ex.: EX/MEM)

- Aceasta funcţionează dacă timpul sursei datelor este anterior timpului de utilizare

- În unele cazuri acest lucru nu se întâmplă şi atunci este nevoie să se introducă **întârzieri** suplimentare (*nop* = no operation sau *bubble* = pipeline stall); aceasta se numeşte *tehnica de întârziere*

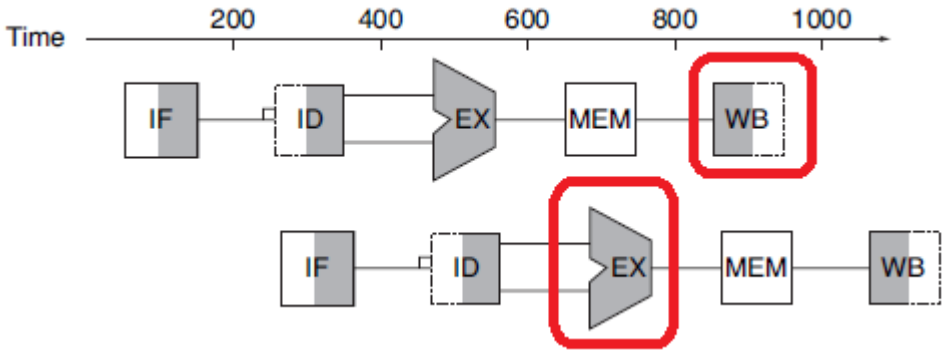
- O tehnică **generală** de evitare a hazardului (de date, dar şi de control) este **reordonarea instrucţiunilor**

Hazard de date

- *Întrebare:* Unde apare hazardul de date ?

```
add $s0, $t1, $t2
sub $t2, $t0, $t3
```

- *Răspuns:* Valorea sumei pentru instr.1 se scrie în registrul `$t0` în etapa WB, în timp ce instr.2 necesită valoarea în etapa EX, care este anterioară ca timp

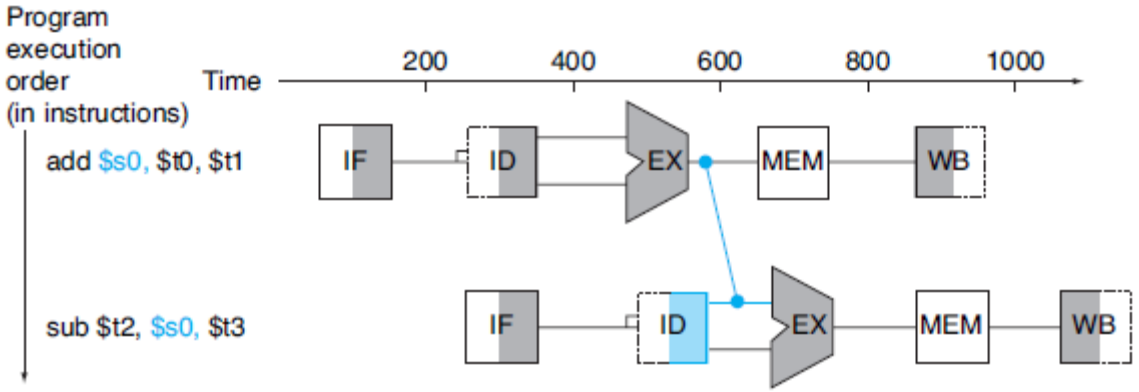


Hazard de date

- *Întrebare:* Cum poate rezolva tehnica de avansare hazardul de date?

```
add $s0, $t1, $t2
sub $t2, $t0, $t3
```

- *Răspuns:* Se preia valoarea lui `$s0` direct după calculul acesteia din instr.1, pentru a putea fi utilizată direct în etapa de execuţie a instr.2



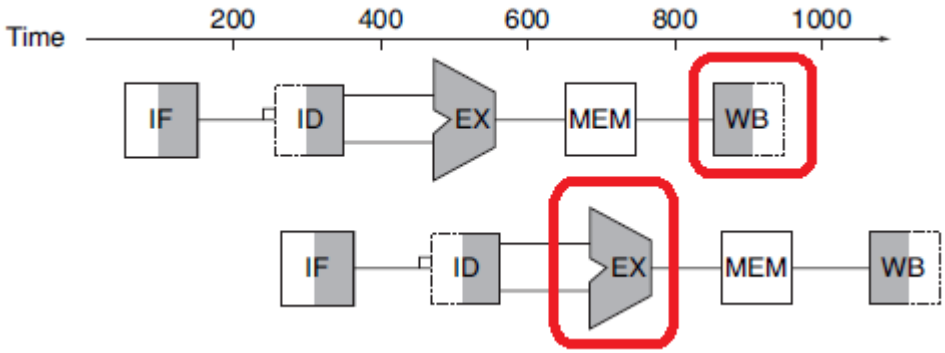
[COD]

Hazard de date

- *Întrebare:* Unde apare hazardul de date ?

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

- *Răspuns:* Valorea încărcată din memorie pentru instr.1 se scrie în registrul `$s0` în etapa WB, în timp ce instr.2 necesită valoarea în etapa EX, care este anterioară ca timp

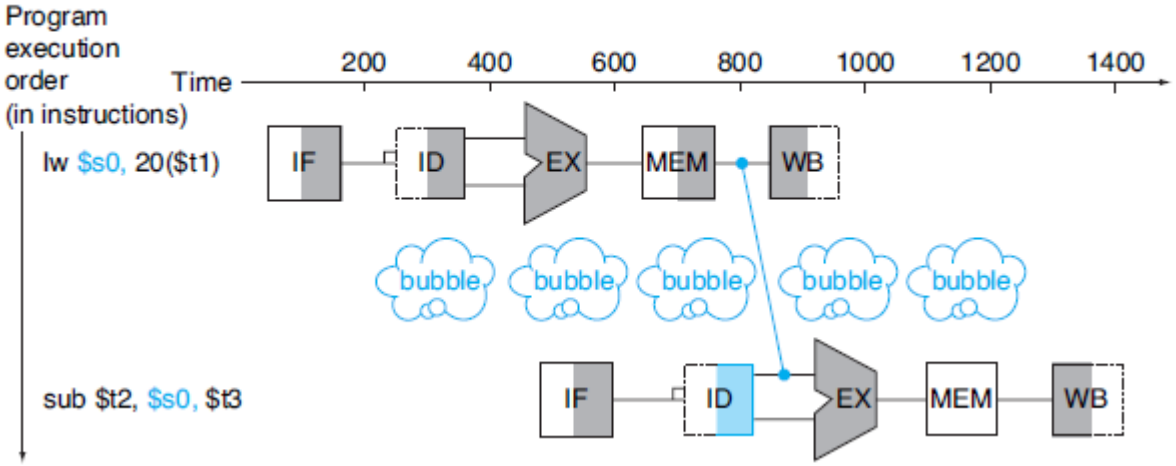


Hazard de date

- *Întrebare:* Cum poate rezolva tehnicile de avansare şi întârziere hazardul de date?

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

- *Răspuns:* Se preia valoarea lui `$s0` direct după citirea din memorie din instr.1, pentru a putea fi utilizată cu 1 singur tact întârziere în etapa de execuţie a instr.2



Hazard de date

- *Întrebare:* Ce face secvenţa de cod de mai jos?

Ne referim la variabilele stocate la locaţiile de memorie astfel:

```
a = 0($t0); b= 4($t0); c = 8($t0); d=12($t0); e=16($t0)
```

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

- *Răspuns:* Calculează:

```
d = a + b
e = c + a
```

Hazard de date

- *Întrebare:* Cum poate rezolva reordonarea codului hazardul de date?
  (care apare spre exemplu la ambele instr. add)

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

- *Răspuns:*

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Hazard de control

- *Hazardul de control* apare când trebuie luată o decizie bazată pe rezultatul unei instrucţiuni în timp ce altele se execută

- Fiindcă nu este introdus de o limitare hardware, este o restricţie **logică**

- În exemplul inţial (spălatul rufelor), apare hazard de control dacă:
  - ✓ se doreşte ajustarea (cantităţii sau a tipului) detergentului folosit în funcţie de cât de bine este spălat un set de rufe

- Pentru procesor, un exemplu de hazard de control apare la condţionări: se încarcă (IF) următoarea instrucţiune, dar nu aceasta este cea care trebuie executată următoarea (din cauza condiţiei de salt care se poate îndeplini sau nu)

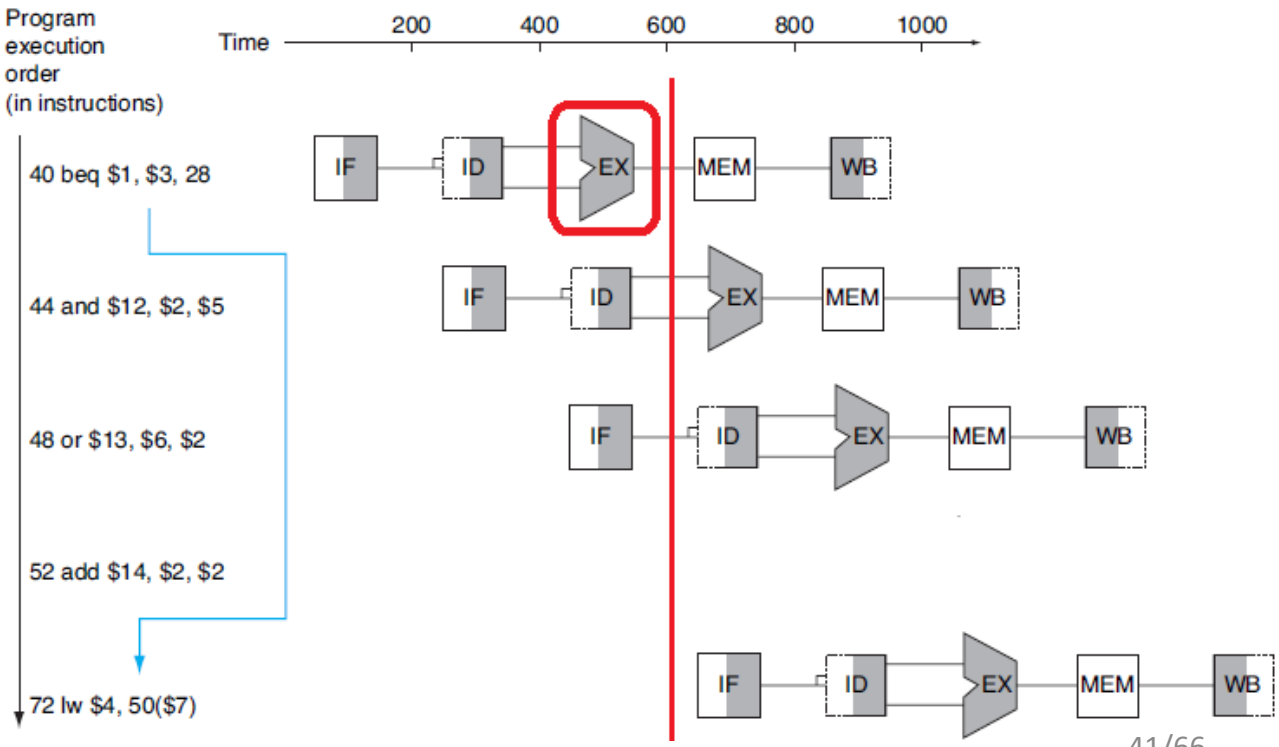
Hazard de control

- *Întrebare:* Unde apare hazardul de control?

```
beq $1, $3, 28
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
lw $4, 50($7)
```

- *Răspuns:* Decizia din instr.1 (salt sau nu) se cunoaşte abia după etapa EX, timp în care se pot încărca în pipeline următoarele instrucţiuni pentru a fi executate

(am presupus salt)



Hazard de control

- O metoda de rezolvare a hazardului de control este *tehnica de întârziere*): se întârzie până când se cunoaşte dacă se realizează salt sau nu, altfel se introduc întârzieri suplimentare (*nop* = no operation sau *bubble* = pipeline stall)

- Pentru a nu aştepta până la aflarea deciziei, se poate utiliza *tehnica de predicţie*: se presupune un anumit rezultat al deciziei şi se continua cu încărcarea în pipeline a instrucţiunii respective

- Se poate lua o decizie statică (ex. pentru un loop mereu se va considera întoarcerea în buclă pentru ca are probabilitate mai mare) sau dinamică (prin contorizarea frecvenţei de apariţie a fiecărei ramuri)

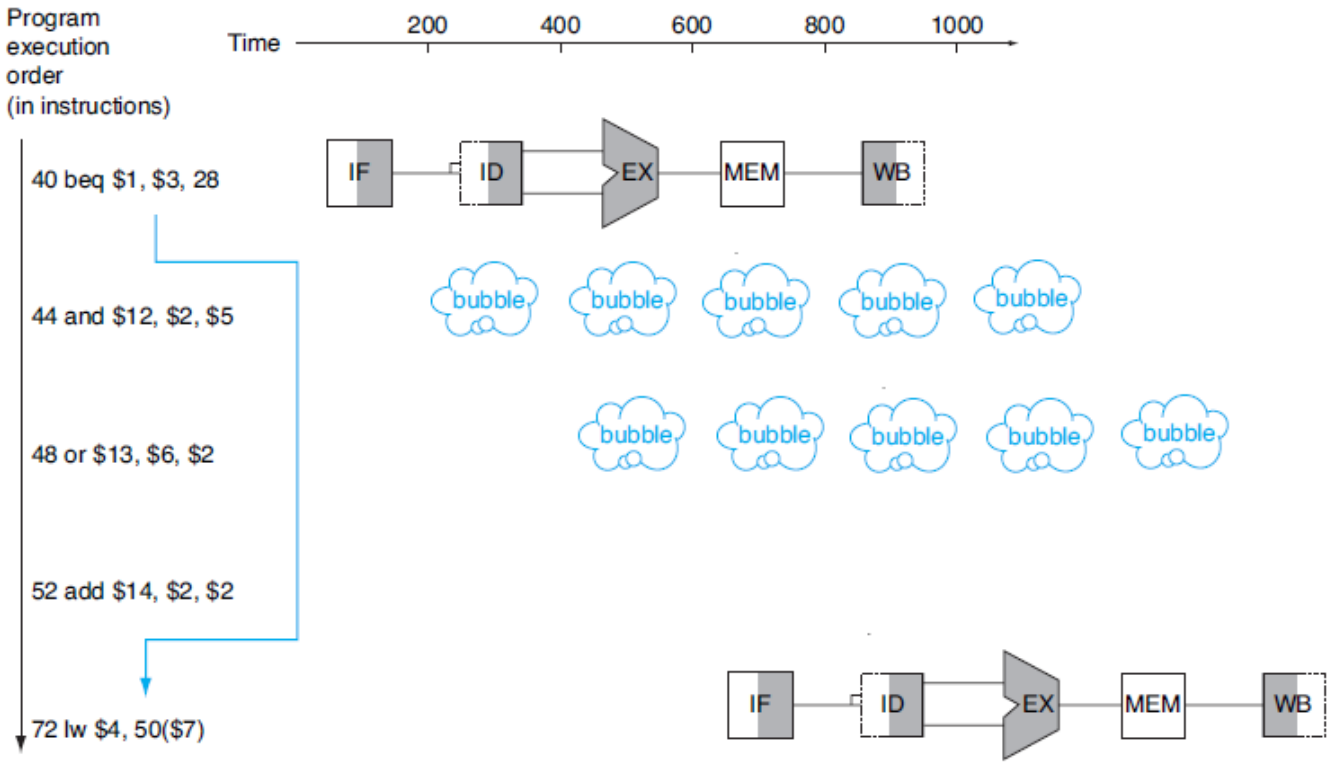
Hazard de control

- *Întrebare:* Cum poate rezolva tehnica de întârziere hazardul de control?

```
beq $1, $3, 28
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
lw $4, 50($7)
```

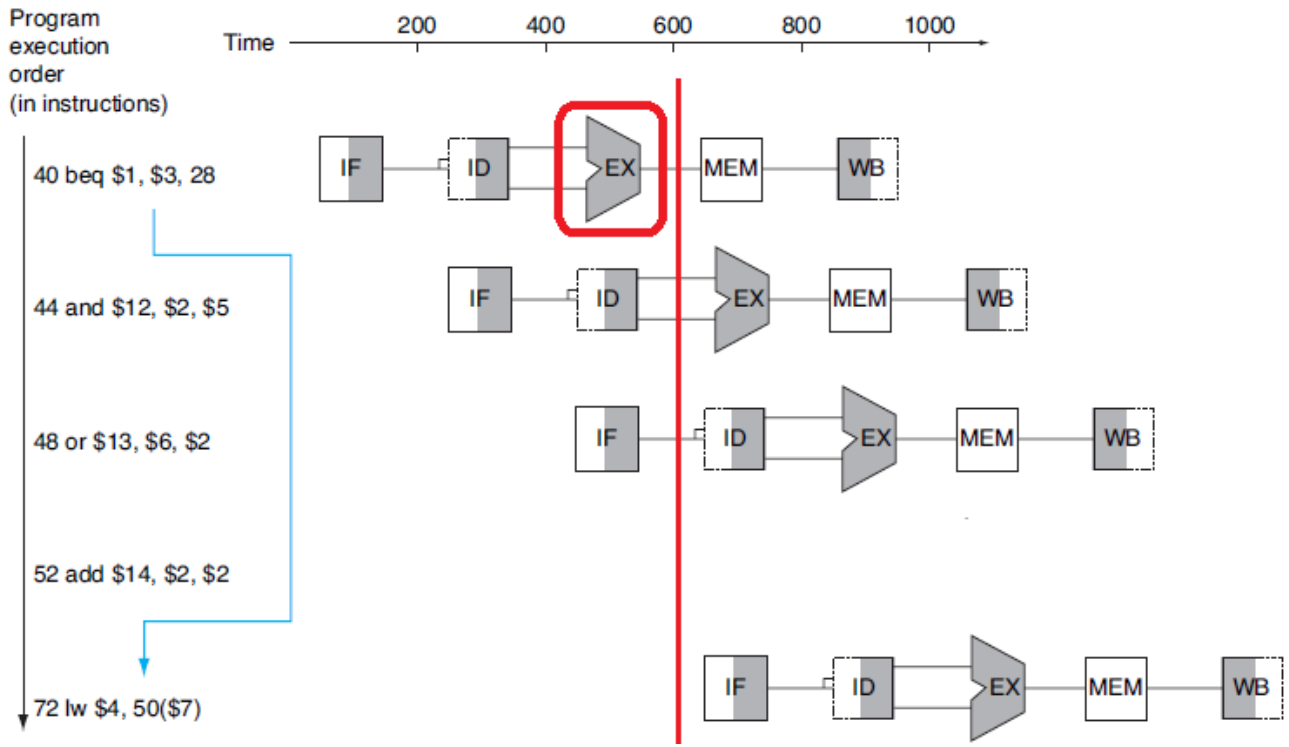
- *Răspuns:*

Se întârzie încărcarea instrucţiunilor până se cunoaşte rezultatul deciziei (am presupus salt)



Hazard de control

- Am folosit deja *tehnica de predicţie*, considerând că nu s-a realizat saltul (predicţie eronată în cazul ilustrat; dar dacă saltul nu se realiza, i.e. predicţia era corectă, nu mai apărea întârzierea de 2 tacturi introdusă prin tehnica întârzierii):



Hazard de control

- Ca să se evite execuţia parţială a instrucţiunilor în caz de predicţii eronate sau întârzierea prin adăugarea bubble / nop, se poate folosi *reordonarea codului*: până se cunoaşte decizia saltului se execută instrucţiuni care se executau indiferent de decizie (situate spre exemplu înainte de instrucţiunea branch)

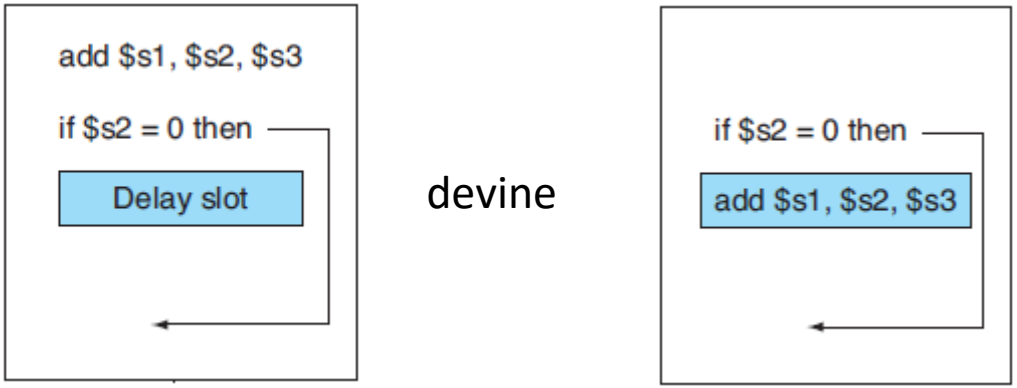
- Tehnica *delayed branch* este utilizată de procesoarele MIPS: se execută întotdeauna instrucţiunea următoare instrucţiunii condiţionate;

- Pentru programator, acest lucru este ascuns, pentru că se reordonează codul şi se obţine acelaşi rezultat.

[Notă: este necesară o singură instrucţiune, nu 2 ca în exemplele precedente, pentru că se poate încărca instrucţiunea conform condiţiei în aceeaşi etapă cu calculul condiţiei de salt]

Hazard de control

- O metoda de rearanjare a codului în caz de delayed branches care este mereu corectă (dar nu se poate realiza întotdeauna):



[Nota: Când nu se poate aplica, se aplică alte metode, care depind de probabilitatea de satisfacere a condiţiei de salt]

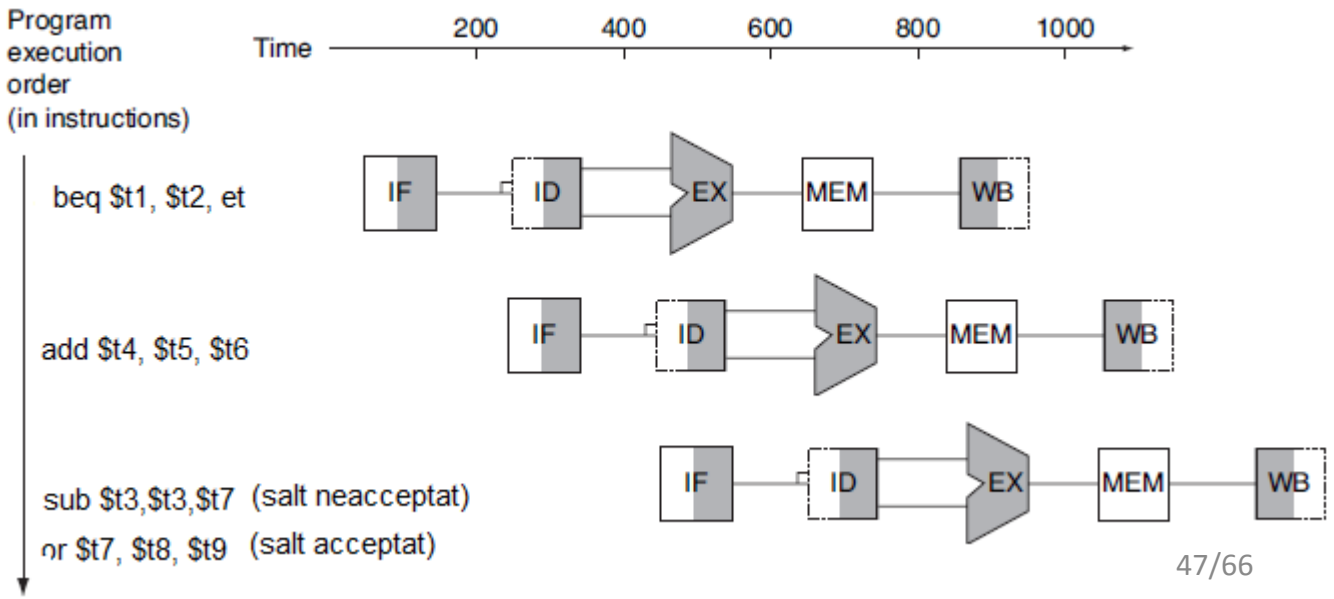
Hazard de control

- *Întrebare:* Cum se execută pe un procesor MIPS secvenţa de instrucţiuni?

```
add $t4, $t5, $t6
beq $t1, $t2, et
sub $t3, $t3, $t7
et:
or $t7, $t8, $t9
```

- *Răspuns:* Instrucţiunea `add` se execută imediat după `beq` (indiferent de rezultatul condiţiei). Am ţinut cont că se încarcă instrucţiunea corespunzătoare în acelaşi tact de ceas cu determinarea condiţiei (tactul 3).

```
beq $t1,$t2,et
add $t4,$t5,$t6
sub $t3,$t3,$t7
et:
or $t7,$t8,$t9
```



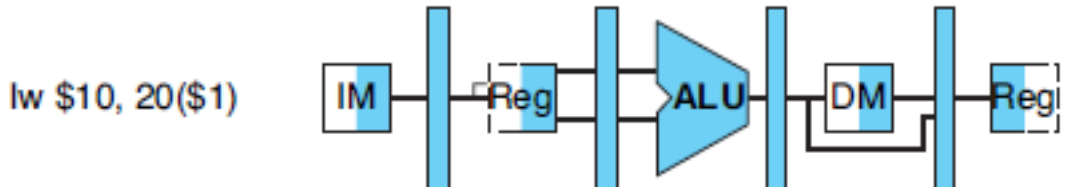
Implementare (III)

- Implementăm schema procesorului cu pipeline, considerând şi hazardul

- Analizăm fiecare dintre tehnicile prezentate:
  - ✓ tehnica de avansare vs. tehnica de întârziere pentru hazardul de date
  - ✓ tehnica de predicţie pentru hazardul de control

Implementare (III)

- Schimbăm puţin reprezentarea instrucţiunilor în reprezentarea cu mai mulţi ciclii simulatan pentru a evidenţia resursele hardware utilizate:

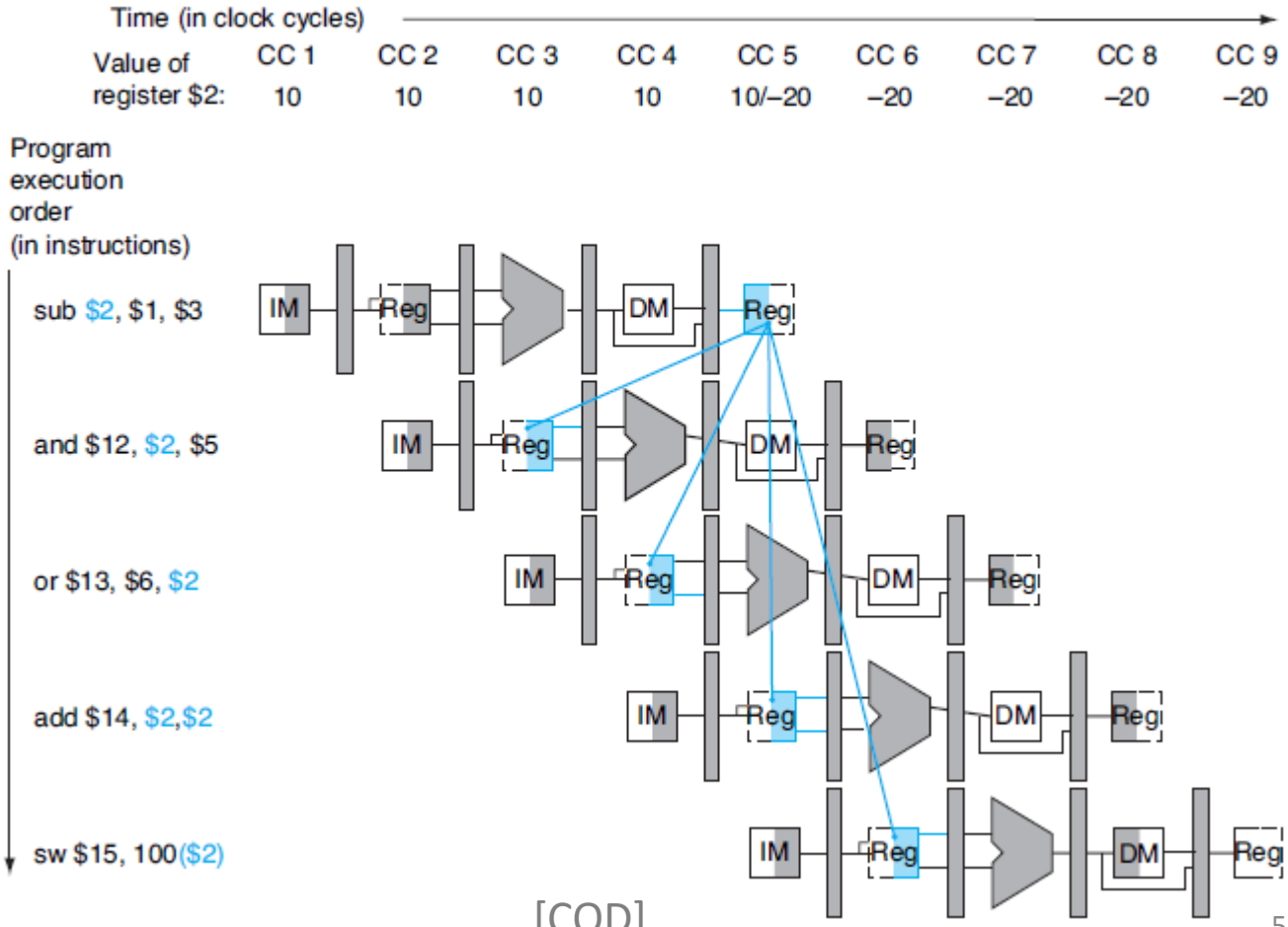


[COD]

- Semnificaţia rămâne aceeaşi: citirea se face pe prima jumătate a tactului şi scrierea în prima jumătate (fiind marcate prin colorare)
- Astfel o valoare poate fi scrisă de o instrucţiune şi citită de alta în acelaşi tact de ceas
- Se introduc regiştrii delimitatori între etapele de pipeline

Tehnica de avansare

- *Întrebare:* Considerăm secvenţa de cod cu dependenţele indicate. Unde apare hazard? Cum poate fi rezolvat prin tehnica de avansare?

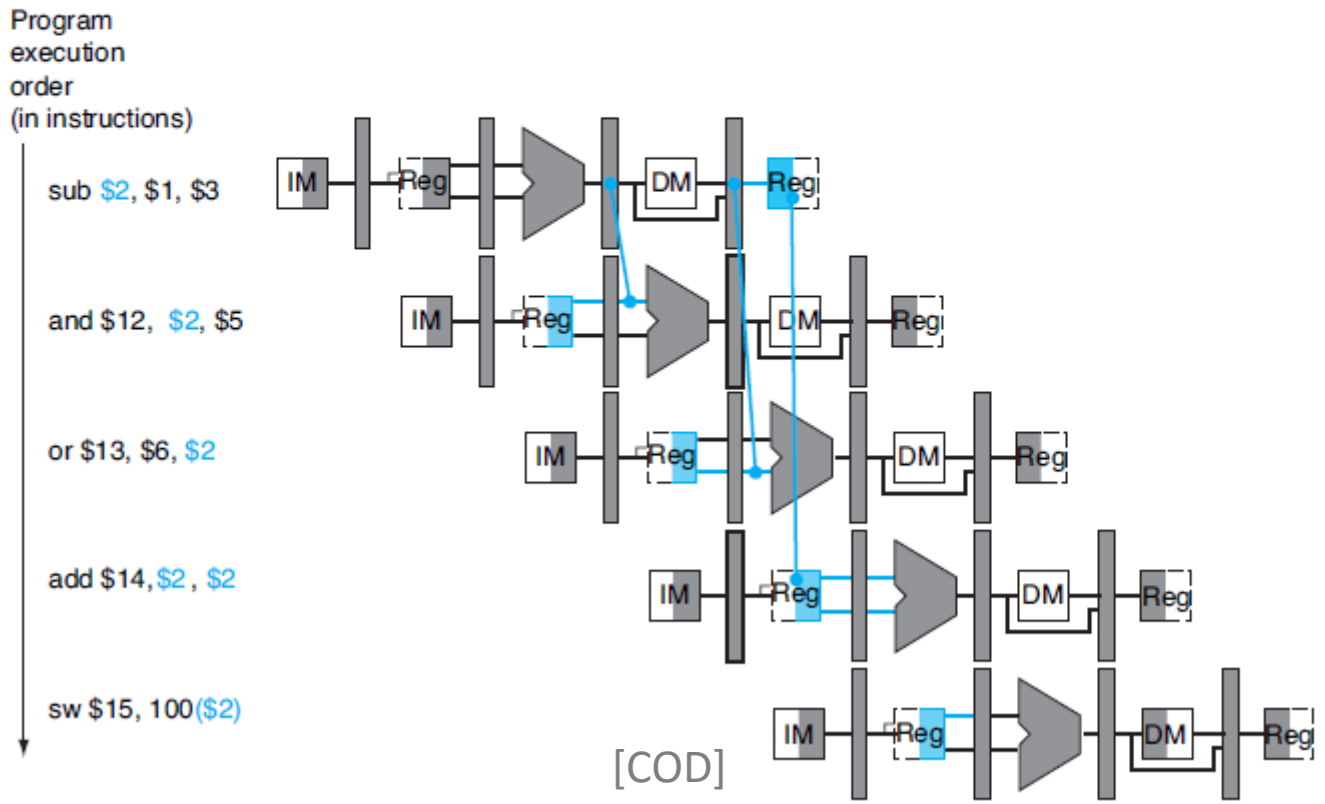


Tehnica de avansare

- *Răspuns :* [Notă: se consideră valoarea 10 în $2 înainte de execuţia instr.1 şi -20 după]

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



Tehnica de avansare

- Informaţia se poate prelua în avans din regiştrii **EX/MEM** (*tip 1 – execuţie*) sau **MEM/WB** (*tip 2 – acces memorie*)

- Notaţia **<reg>.<câmp>** semnifică câmpul (semnalul de control, registrii sursă sau destinaţie, etc.) din registrul pipeline <reg>; Spre exemplu, **EX/MEM.RegisterRd** este registrul destinaţie (registrul 3 din formatul instrucţiunii) care se găseşte în registrul pipeline EX/MEM

- Astfel, cele2 tipuri de avansări sunt:
  - ✓ **Tipul 1** – un registru sursă (rs, rt) trebuie preluat din EX/MEM (după execuţie):

    (1a) EX/MEM.RegisterRd = ID/EX.RegisterRs

    (1b) EX/MEM.RegisterRd = ID/EX.RegisterRt
  - ✓ **Tipul 2** – un registru sursă (rs, rt) trebuie preluat din MEM/WB (după acces.mem.):

    (2a) MEM/WB.RegisterRd = ID/EX.RegisterRs

    (2b) MEM/WB.RegisterRd = ID/EX.RegisterRt

Tehnica de avansare

- *Întrebare:* De ce tip este primul hazard din exemplul anterior (`and`)? Dar cel de-al doilea (`or`)?

- *Răspuns:*

Pentru instrucţiunea `add`, $2 este primul registru sursă (ID/EX.RegisterRs), care se preia din EX/MEM a instrucţiunii sub (EX/MEM.RegisterRd), deci este cazul 1a.

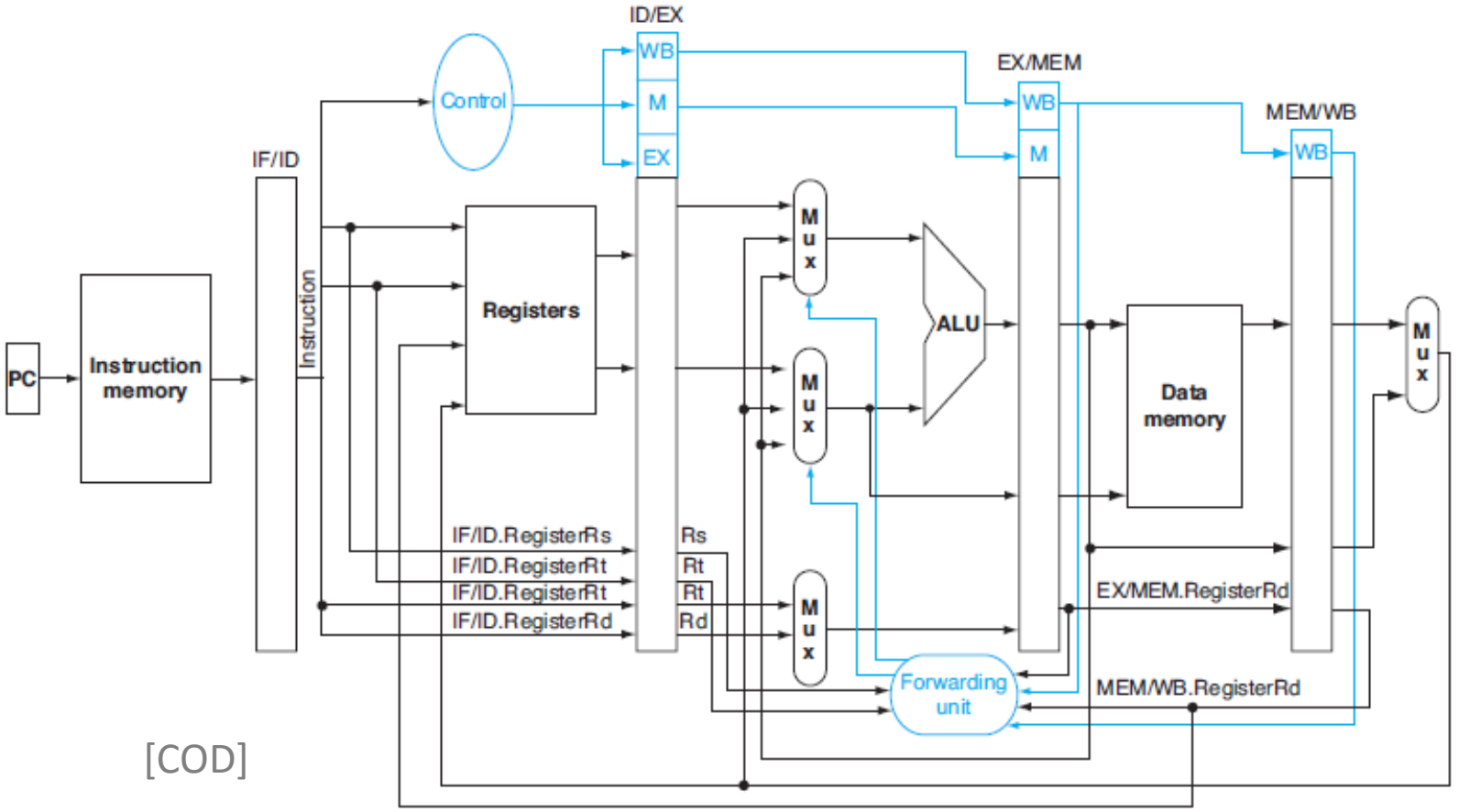
Pentru instrucţiunea `or`, $2 este al doilea registru sursă (ID/EX.RegisterRt), care se preia din MEM/WB a instrucţiunii sub (MEM/WB.RegisterRd), deci este cazul 2b.

- Atenţie, pot să apară mai multe (sub)tipuri de hazard simultan!

Tehnica de avansare

- Introducem deci unitatea de avansare (*forwarding unit*), care comanda preluarea datelor din regiştrii EX/MEM sau MEM/WB

  [Notă: se reprezintă doar calea de date, considerand doar instr. in format R: add, sub, and, or]



[COD]

Tehnica de avansare

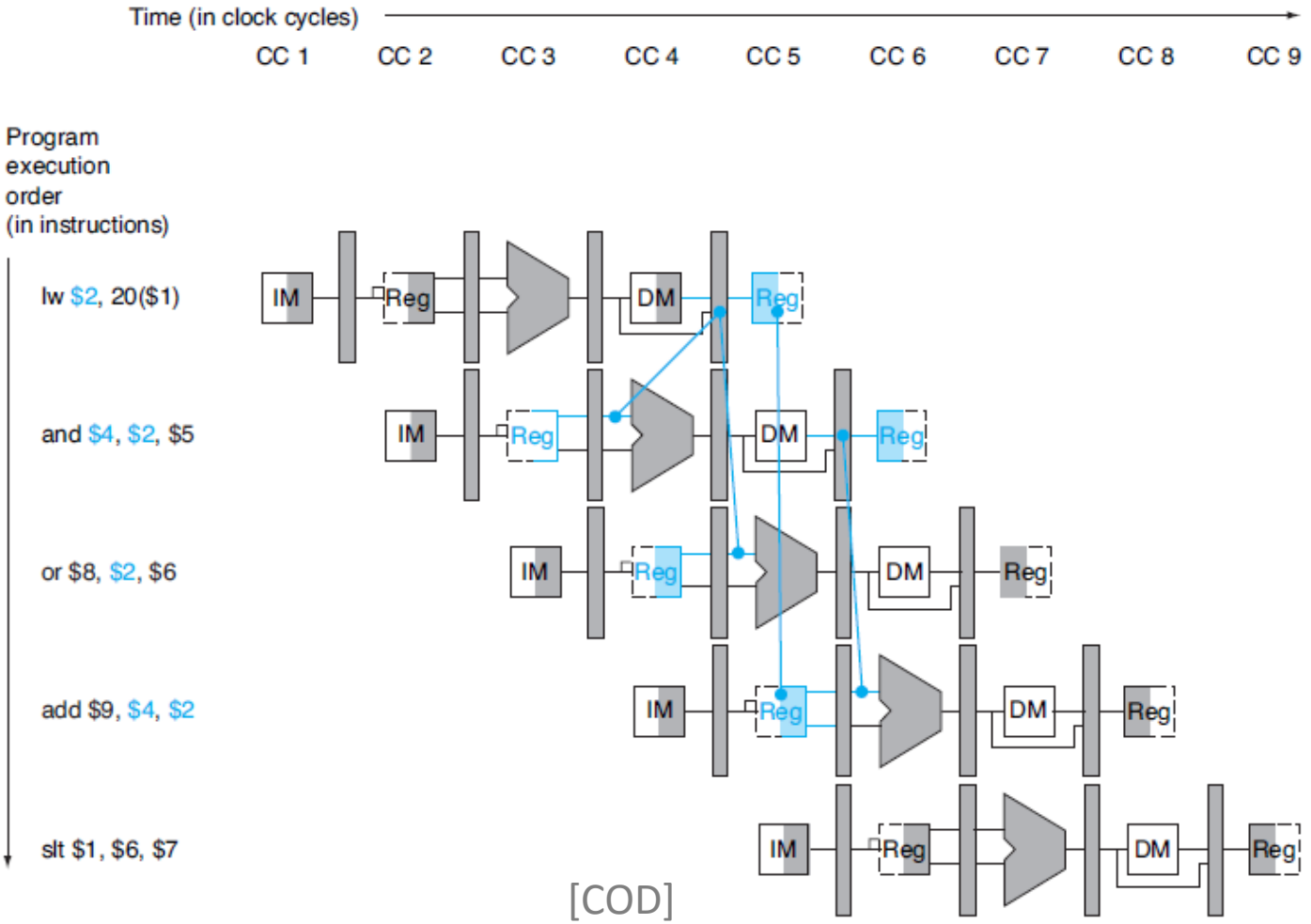
- Valorile scoase la ieşire de unitatea de control *forwarding unit* :

ForwardA / ForwardB	Registru pipeline sursa	Semnificatie
00	ID/EX	Primul / al doilea operand ALU provine din fisierul de registrii (register file)
10	EX/MEM	Primul / al doilea operand ALU provine din rezultatul ALU anterior
01	MEM/WB	Primul / al doilea operand ALU provine din memoria de date sau un rezultat ALU anterior (propagat prin reg. pipeline)

[Notă: nu detaliem construcţia Forwarding Unit]

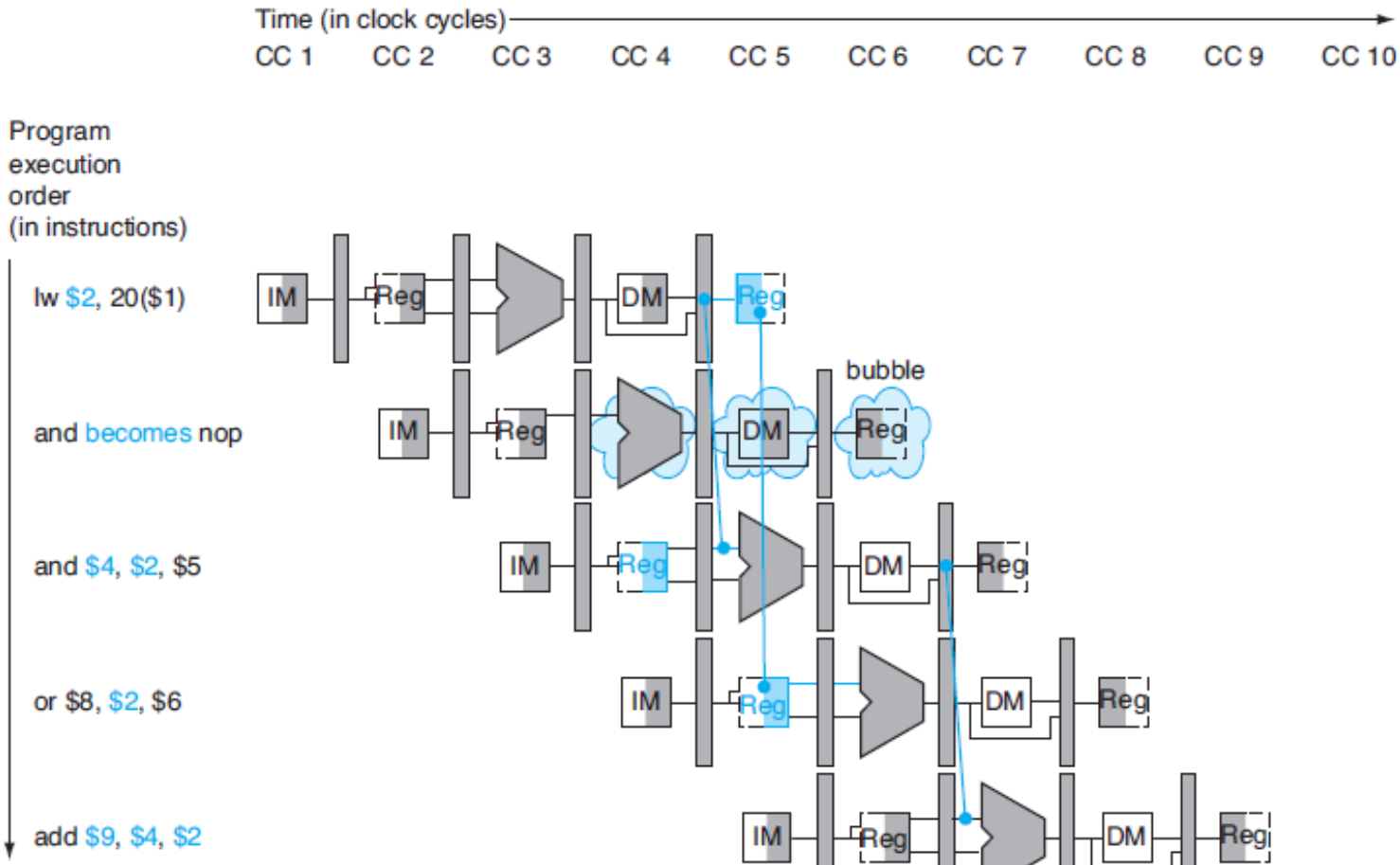
Tehnica de întârziere

- *Întrebare:* Considerăm secvenţa de cod cu dependenţele indicate. Unde apare hazard? Cum poate fi rezolvat prin tehnica de întârziere?



Tehnica de predicţie

- *Răspuns :*



Tehnica de întârziere

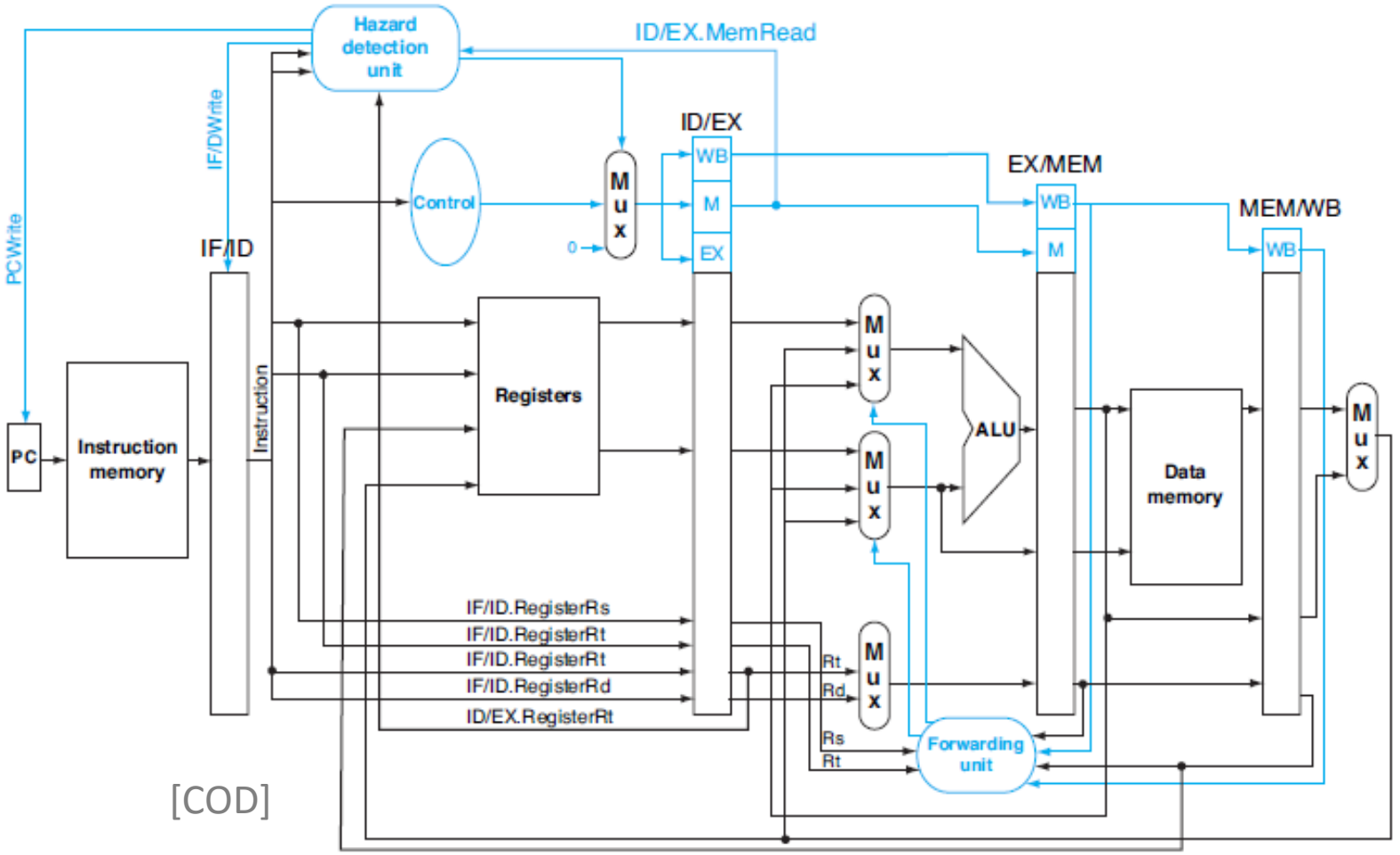
- Întârzierea trebuie să simuleze o instrucţiune *nop* (no operation) care să înceapă din faza EX (acolo unde s-a realizat hazardul)

- Întârzierea nu estre realizată exact ca în figura anterioară, ci:
  - ✓ Stagiile IF şi ID pentru instrucţiunea add se realizează de fapt în CC2, respectiv CC3; etapa EX este întârziată până în CC5
  - ✓ etapa IF a instrucţiunii or se realizează în CC3, dar ID se întârzie până în CC5

- În general, o unitate de detecţie al hazardului permite:
  - ✓ folosirea aceloraşi date în fazele IF şi ID prin desetarea semnalelor PCWrite, IF/IDWrite
  - ✓ desetarea semnalelor de control în fazele EX, MEM, WB

Tehnica de întârziere

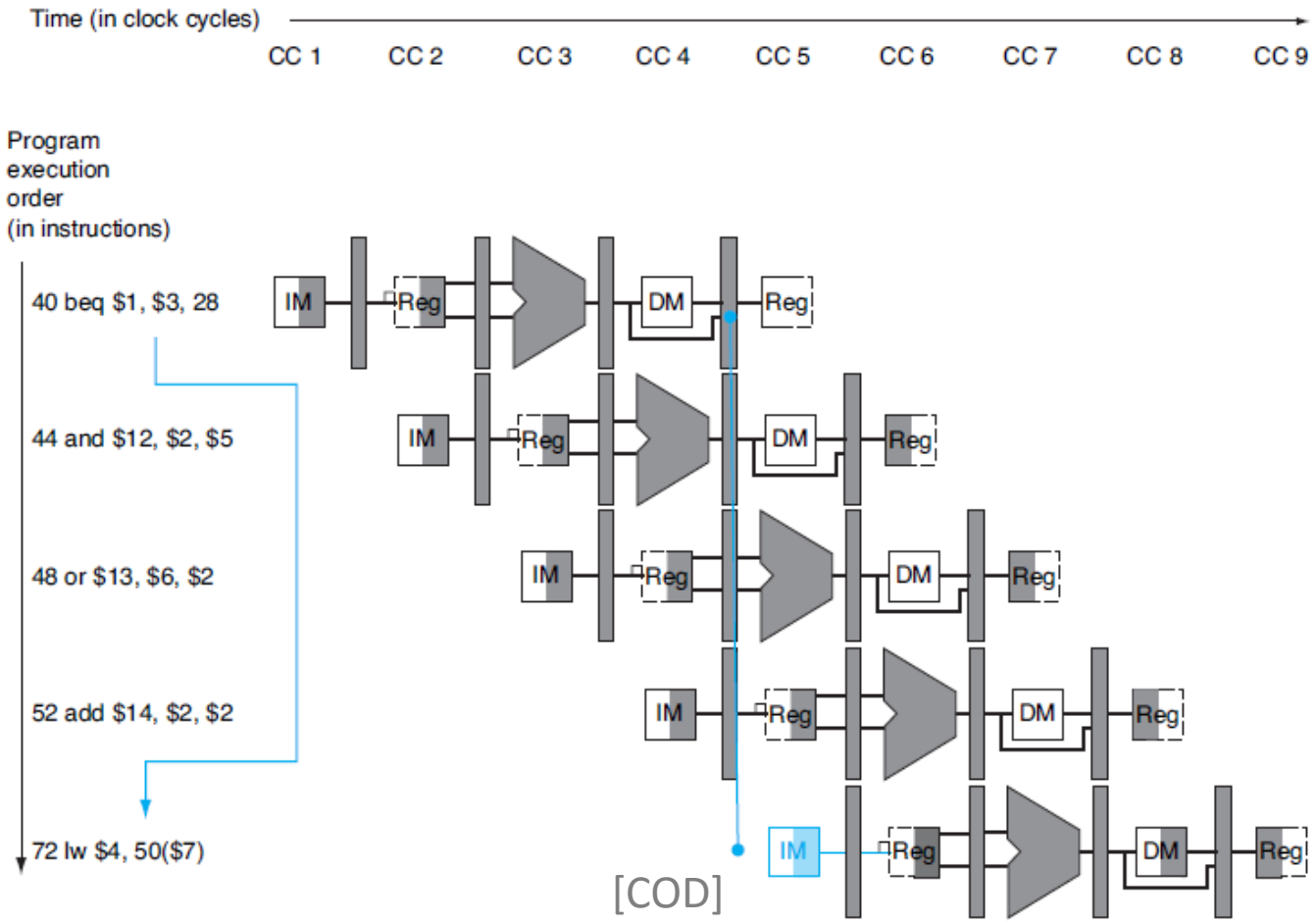
- Cu unitatea de detecţie a hazardului (*hazard detecting unit*), schema devine:



[COD]

Tehnica de predicţie

- Considerăm că nu se realizează saltul, deci instrucţiunile se introduc secvenţial in pipeline



Tehnica de predicţie

- Dacă se realizează saltul, instrucDacă nu se realizează saltul, atunci se continuă (prezumpţia a fost adevărată)

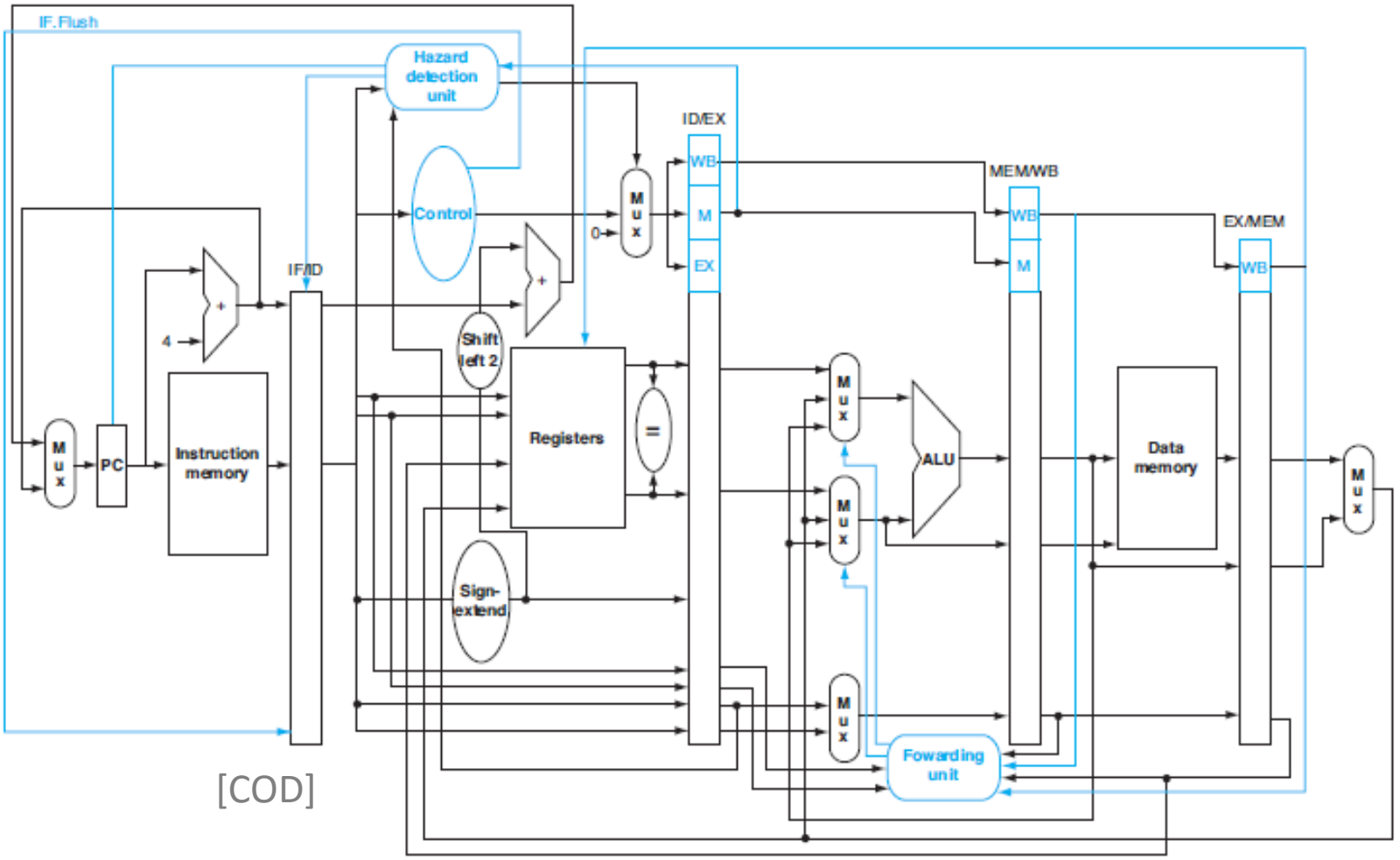
- Dacă se realizează saltul, instrucţiunile care nu ar fi trebuit introduse în pipeline se elimină (*flush*)

- Eliminarea unei instrucţiuni înseamnă:
  - ✓ Resetarea semnalelor de control la 0 (ca şi în cazul staţionării)
  - ✓ Ştergerea instrucţiunii din regiştrii de pipeline, adică setarea 0 (*nop*) în IF/ID pentru că instrucţiunea următoare saltului se găseşte în această etapă

- În cazul beq, testarea egalităţii se poate face mai simplu (nu cu scădere ALU), ci prin testare bit-cu-bit a valorii din cei 2 regiştrii, operaţie care poate fi făcută în etapa ID (şi afişată pe schemă)

Tehnica de întârziere

- În final, se obţine:



[COD]

Excepţii

- Excepţiile sunt evenimente neprogramate care întrerup execuţia programului

- Întreruperile sunt excepţii care provin din afara procesorului (anumite arhitecturi se referă la întreruperi cu aceeaşi semnificaţie ca şi excepţiile)

- Regiştrii specifici pentru tratarea excepţiilor:
  - ✓ *EPC* (*E*xception *P*rogram *C*ounter): stochează adresa instrucţiunii care a generat excepţia (sau pe cea următoare)
  - ✓ *Cause*: stochează cauza excepţiei (ex.: 10 = instrucţiune necunoscută; 12 = arithmetic overflow)

- În continuare, considerăm doar excepţiile de tip arithmetic overflow, cu adresa procedurii pentru excepţii la 0x 8000 0180

Excepţii

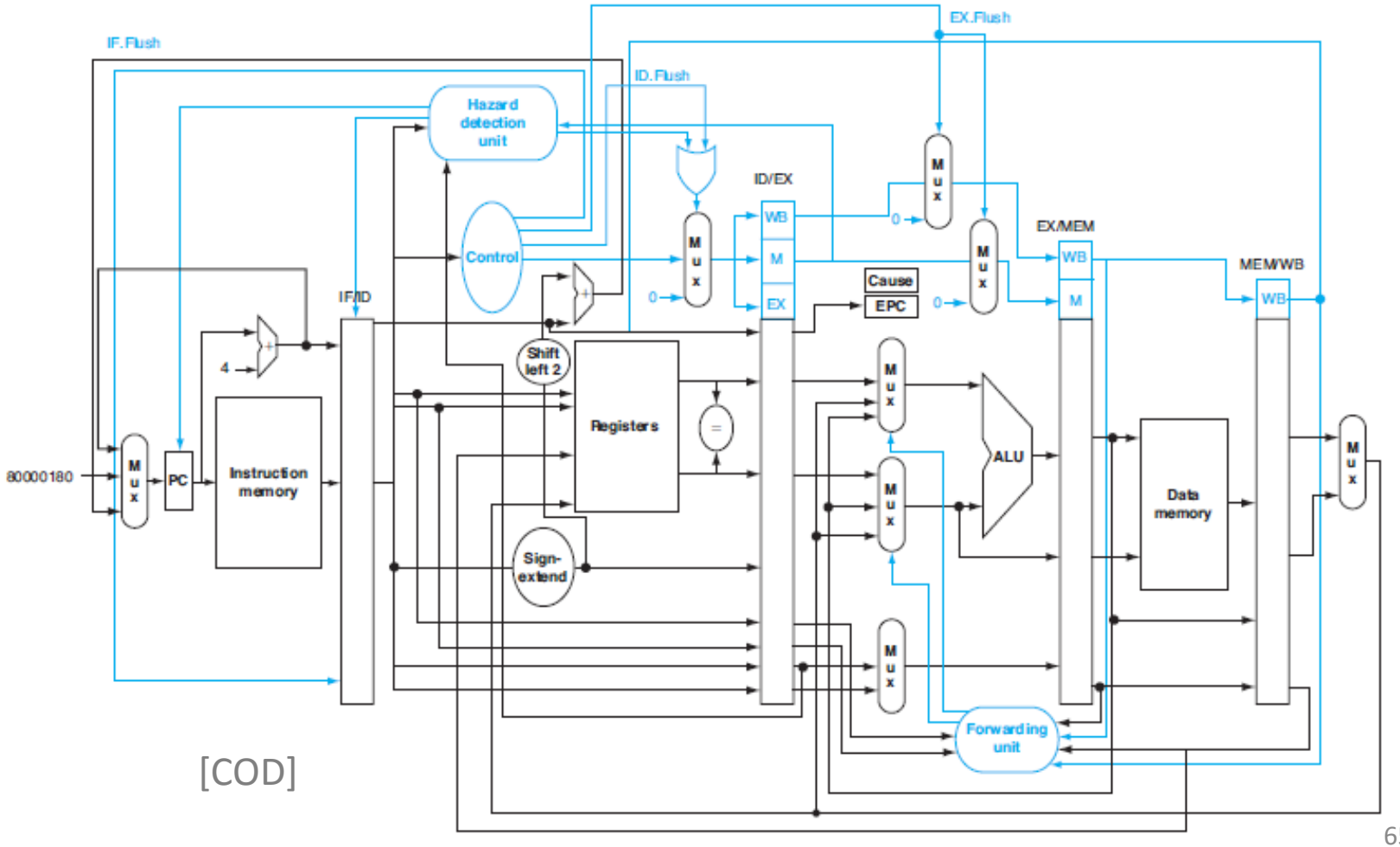
- În pipeline, excepţia de arithmetic overflow se tratează asemănător unui hazard de control:
  - ✓ se curăţă (flush) instrucţiunea care urmează instrucţiunii care a generat overflow
  - ✓ se execută instrucţiunile începând cu adresa procedurii de excepţie (0x 8000 0180)

- Curăţarea instrucţiunii în etapa ID se face printr-un semnat de control *ID.Flush* (care intră într-o poartă OR cu semnalul de flush al unităţii de detecţie a hazardului)

- Curăţarea instrucţiunii în etapa EX se face printr-un semnat de control *EX.Flush* (care intră într-o poartă OR cu semnalul de flush al unităţii de detecţie a hazardului)

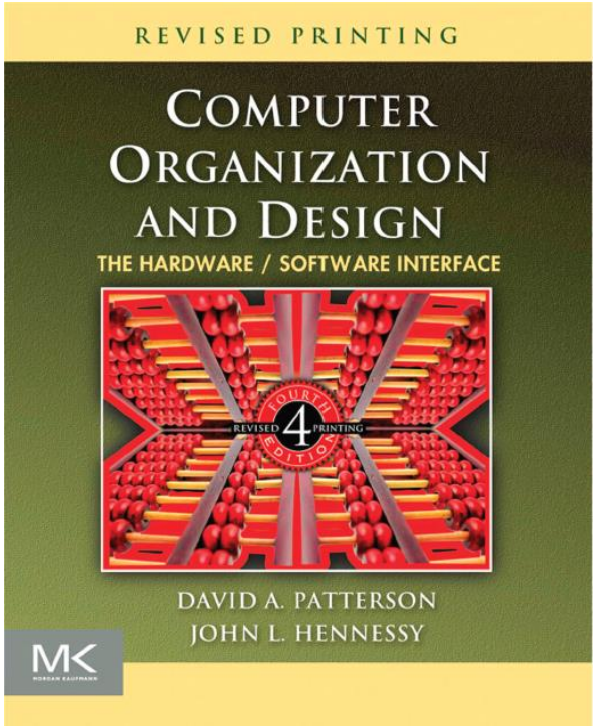
Implementare (III)

- Schema (simplificată) a procesorului care implementează pipeline cu hazard şi excepţie de overflow:



[COD]

Referințe bibliografice



[COD] D. Patterson and J. Hennessy, Computer Organisation and Design

[AAT] A. Atanasiu, Arhitectura calculatorului

Schemele [Xilinx - ISE] au fost realizate folosind
http://www.xilinx.com/tools/projnav.htm

Grafurile [JFLAP] au fost realizate folosind
http://www.jflap.org/