Limbajul mașinii

Principii de proiectare

Introducere

- Limbajul unui calculator = cuvinte + vocabular

  - Cuvintele -> instrucțiuni
  - Vocabularul -> set de instrucțiuni

Operaţiile hardware-ului

- Orice calculator trebuie să fie capabil să efectueze operaţii aritmetice: add a, b, c
- Exemplu: a = b + c + d + e
  - add a, b, c
  - add a, a, d
  - add a, a, e

  CONCLUZIE: Este nevoie de 3 instrucţiuni
- Hardware-ul pentru implementarea unei operaţii cu 3 operanzi este mai simplu decât hardware-ul necesar implementării unei operaţii cu un număr variabil de operanzi

- PRINCIPIUL DE PROIECTARE 1 - Simplitatea favorizează uniformitatea

Exerciţiu: Se dă următoarea comandă C:

f = (g+h) - (I+j)

Ce cod va produce un compilator C ?

Operaţiile hardware-ului

- Orice calculator trebuie să fie capabil să efectueze operaţii aritmetice: add a, b, c
- Exemplu: a = b + c + d + e
  - add a, b, c
  - add a, a, d
  - add a, a, e

  CONCLUZIE: Este nevoie de 3 instrucţiuni
- Hardware-ul pentru implementarea unei operaţii cu 3 operanzi este mai simplu decât hardware-ul necesar implementării unei operaţii cu un număr variabil de operanzi

- PRINCIPIUL DE PROIECTARE 1 - Simplitatea favorizează uniformitatea

Exerciţiu: Se dă următoarea comandă C:

f = (g+h) - (I+j)

Ce cod va produce un compilator C ?

Operanzii hardware-ului

- Operanzii instrucţiunilor aritmetice provin dintr-un număr limitat de locaţii speciale denumite **registre**
- Dimensiunea unui registru este de 32 biţi => un cuvânt va avea 32 de biţi
- Numărul de registre este limitat într-un calculator - 32 de registre generale
- PRINCIPIUL DE PROIECTARE 2 - Mai mic înseamnă mai rapid

Notaţii: $s0, $s1 … registre folosite pentru variabile; $t0, $t1 … registre temporare necesare compilării unui program în instrucţiuni MIPS

Exerciţiu: Se dă următoarea comandă C:

f = (g+h) - (I+j)

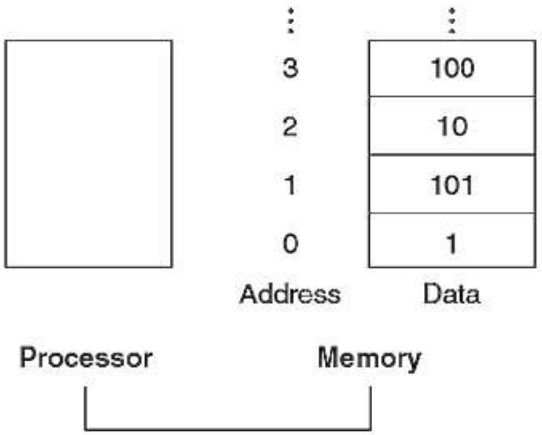
Ce cod va produce un compilator C ?

Soluţie:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
add $s0, $t0, $t1
```

- Structurile de date foarte mari: matricile şi vectorii sunt ţinute în păstrate în memorie => necesitatea existenţei instrucţiunilor care să realizez transferul datelor între memorie şi registre

  INSTRUCŢIUNI DE TRANFER DE DATE



- Instrucţiunea care deplasează datele din memorie într-un registru este numită LOAD

- Formatul instrucţiunii:
  - Numele operaţiei
  - Registrul ce trebuie încărcat
  - Constantă
  - Registrul folosit pentru accesarea memoriei

- **Exerciţiu**: A este un tablou de 100 de cuvinte, iar variabilele g şi h sunt asociate cu registrele $s1 şi $s2. Adresa de bază este în $s3. Cum se translatează g=h+A[8] ?

- Compilatorul alocă structurile de date locaţiilor din memorie, apoi pună adresa corectă de început în instrucţiunea de transfer de date.

- Cuvintele secvenţiale diferă prin 4. Cuvântul în MIPS trebuie să înceapă la o adresă care este multiplu de 4: RESTRICŢIE DE ALINIERE

- Calculatoare pot folosi adresele celui mai din stânga octet - BIG END - ca adresă a cuvântului, sau folosesc octetul cel mai din dreapta - LITTLE END

- Adresarea la nivel de octet afectează din nou indexarea tablourilor => offset-ul care trebuie adăugat registrului de bază $s3 trebuie să fie 4x8=32 biţi pentru a selecta A[8] şi nu A[8/4]

- Instrucţiunea complementară intsrucţiunii LOAD este instrucţiunea STORE (memorează)

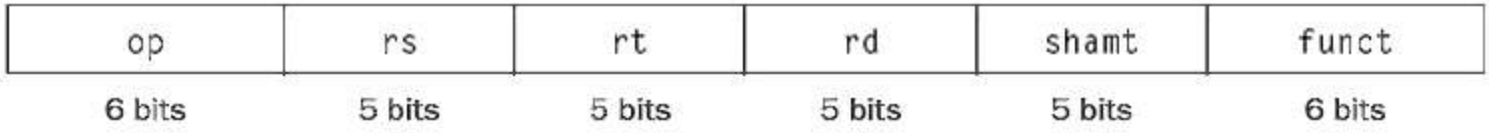
- Formatul instrucţiunii STORE este următorul:
  - Nume operaţie
  - Registrul de memorat
  - Offset-ul pentru selectarea elementului din tablou
  - Registrul de bază

- **Exerciţiu**: Variabila h este asociată cu registrul $s2 şI adresa de bază a tabloului A este în $s3. Detaliaţi A[12] = h + A[8]

- Memoriile actuale sunt foarte mari => adresa de bază a unui tablou este introdusă într-un registru deoarece n u este loc suficient în zona rezervată pentru offset.

Reprezentarea instrucţiunilor

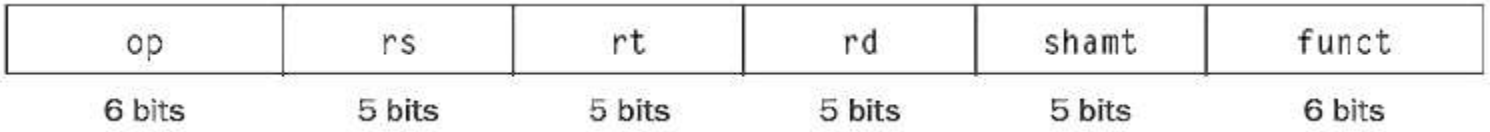
Instrucţiunile sunt păstrate în calculator ca o serie de semnale electronice înalte sau joase => pot fi reprezentate ca numere



op	OPCODE
rs	registru sursă pentru primul operand
rt	registru sursă pentru al doilea operand
rd	registrul operandului de destinaţie ce primeşte rezultatul operaţiei
shamt	numărul deplasărilor
funct	selectează varianta specifică a operaţiei din câmpul op

- PRINCIPIUL DE PROIECTARE 3 - Proiectarea bună necesită compromisuri bune

- Instrucţiunea de tip R



- Instrucţiunea de tip I

- Folosirea mai multor tipuri de instrucţiuni => complicarea hardware-ului, dar putem reduce complexitatea dacă păstrăm formate asemănătoare

- PRINCIPII DE BAZĂ
  - Instrucţiunile sunt reprezentate ca numere
  - Programele pot fi păstrate în memorie pentru a putea fi citite/scrise precum numerele

INSTRUCŢIUNI DE DECIZIE

- Ramificaţii condiţionale

  beq registru_1, registru_2, L1 #dacă registru_1 = registr_2 dute la eticheta L1

  bne registru_1, registru_2, L1 # dacă registru_1 ≠ registru_2 dute la eticheta L1

- Asamblorul este cel care calculează adresele pentru ramificaţii, calculează adresele instrucţiunilor de încărcare/memorare

- **Exerciţiu**: Folosind registrele $s0 … $s4 pentru cele 5 variabile de mai jos, să se detalieze codul generat pentru
  - If (i==j) f=g+h; else f=g-h

Proceduri

- Execuţia unei proceduri presupune următorii paşi

- Pune parametrii într-un loc unde pot fi accesaţi de către procedură
- Transferă controlul procedurii
- Obţine resursele de memorie necesare procedurii
- Efectuare sarcină
- Pune valoarea rezultată într-un loc de unde poate fi accesată de către programul apelant
- Întoarce controlul la punctul de plecare

Registrele folosite:

$a0 - $a3	patru registre de argumente în care se comunică parametrii
$v0 - $v1	două registre de valori în care să întoarcă rezultatul
$ra	un registru al adresei de întoarcere

Folosirea mai multor registre

- Compilatorul are nevoie pentru o procedură de mai mult de 4 registre pentru argumente şi 2 pentru întoarcerea rezultatului.

- Structura de date ideală pentru transferul în memorie al registrelor este stiva

$sp - pointer-ul stivei

PUSH

POP

Proceduri imbricate

- Procedura A are in teriorul ei procedura B sau cazurile de recursivitate

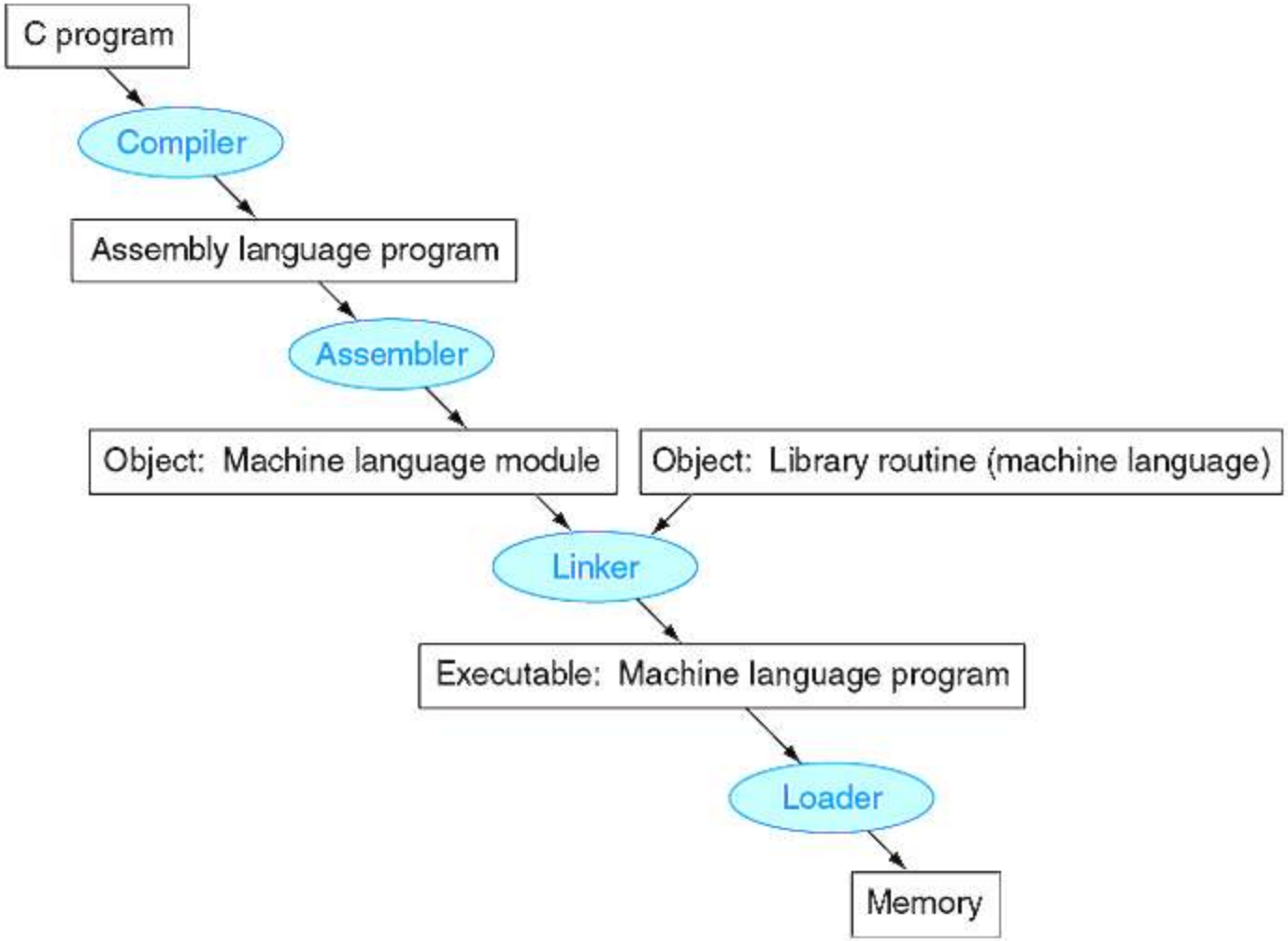
- Introducem în stivă toate registrele care trebuie păstrate după apel. Apelantul introduce în stivă registrul adresei de întoarcere $ra şi orice registru salvat folosit de apelant.

- Pointer-ul stivei $sp este corectat pentru a lua în evidenţănumărul registrelor puse în stivă. La întoarcere, registrele sunt refăcute din memorie, iar pointer-ul stivei este ajutat.

Modurile de adresare MIPS

- MIPS are următoarele moduri de adresare
    - Adresare a registrelor
    - Adresare prin bază sau deplasare
    - Adresare imediată
    - Adresare relativă la PC
    - Adresare pseudodirectă

Cei 4 pași necesari pentru conversia unui program

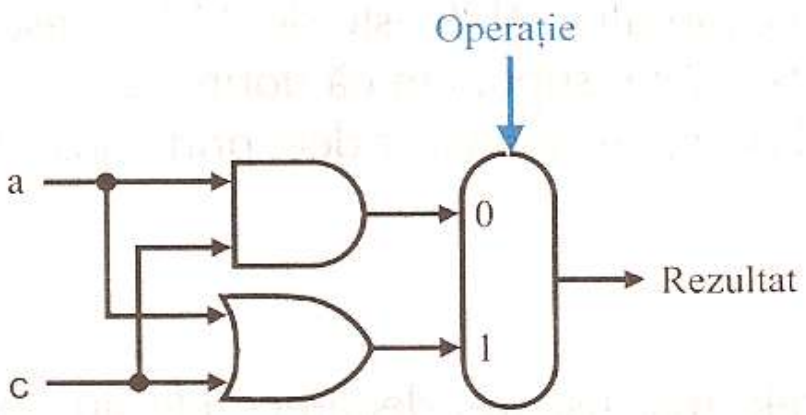


Proiectarea UAL

Operații logice cu corespondență directă în hardware

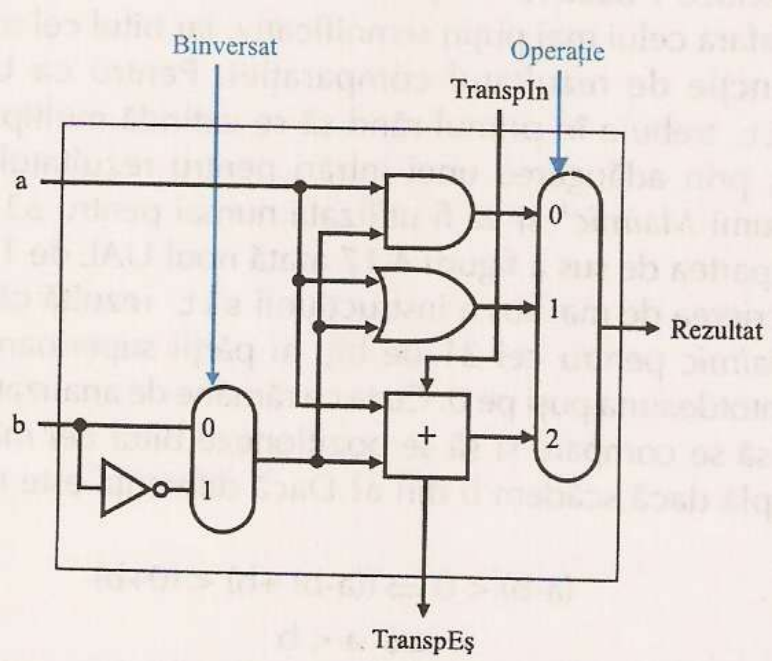
AND	c = a * b
SAU	c = a + b
INV	c = not a
MUX	dacă d==0, c=a altfel c=b

Unitatea logică pentru 1 bit – liniile albastre sunt linii de comandă



Realizarea operației de adunare presupune existența unui sumator complet

Realizarea operației de scădere presupune introducerea unui inversor

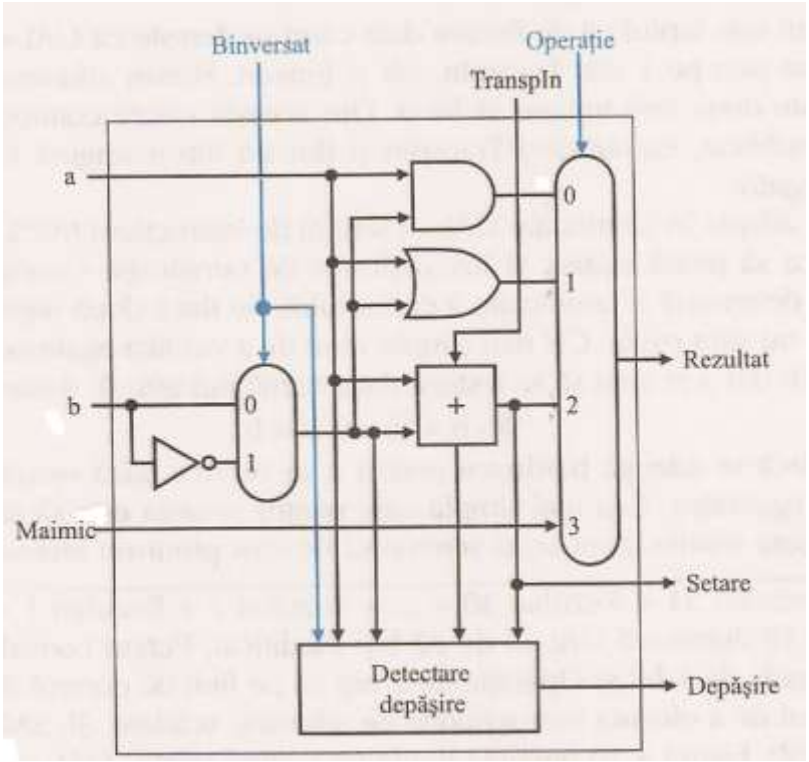


Hardware-ul rezultat este foarte simplu => cc2 standard universal pentru aritmetica numerelor întregi

Ce operație încă nu se poate realiza ?

Operația ***slt*** produce 1 dacă rs < rt şi 0 altfel

Se observă că este suficient să determinăm bitul de semn al operației a-b



Cum se poate modifica figura alăturată astfel încât să implementăm ramificarea condiţională ?

Sumatorul este implementat cu ajutorul unui CLA - Carry Look Ahead prezentat la CN 1

Operaţiile de înmulţire/împărţire inclusiv virgula mobilă au prezentate în CN 1

Calea de date şi de control

Durata perioadei de ceas precum şi numărul de cicluri/instrucţiune sunt date de implementarea procesorului.

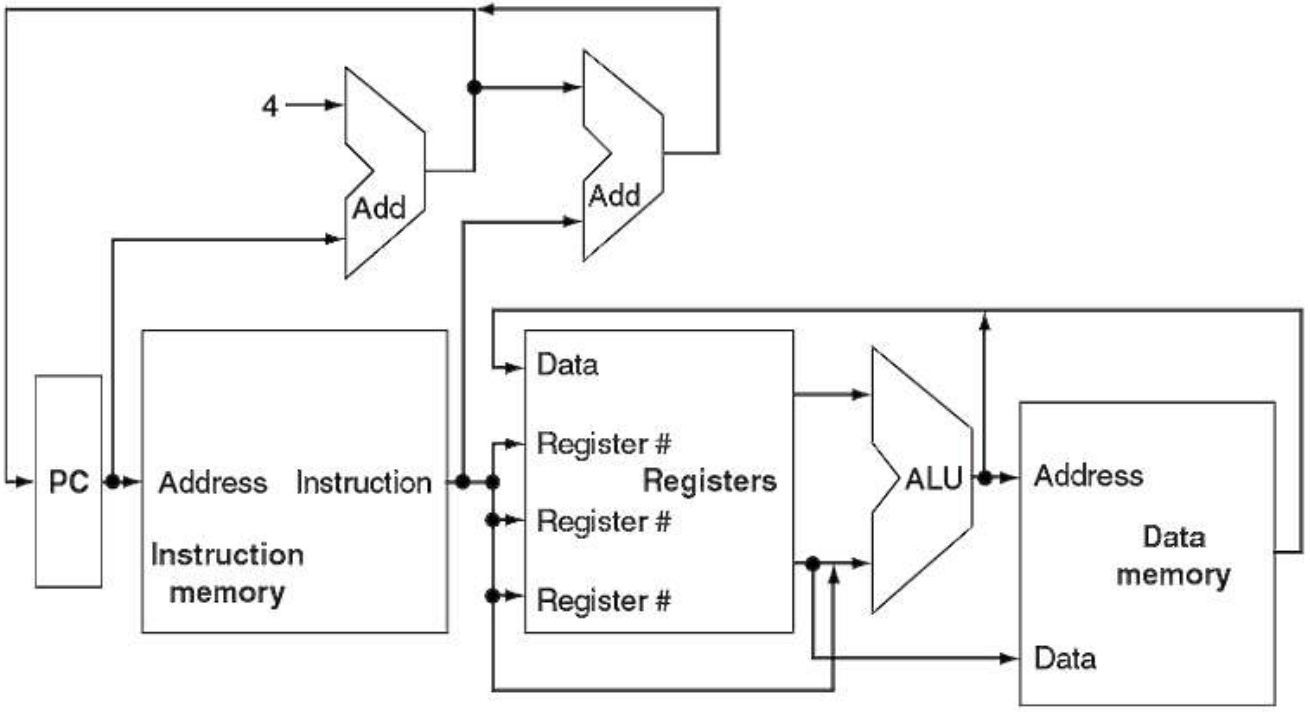
Ne propunem implementarea următoarelor instrucţiuni

- ✓ instrucţiuni de referire a memoriei - încărcare/memorare
- ✓Instrucţiuni aritmetice şi logice - add, sub, and, or şi slt
- ✓Instrucţiunile de ramificaţie la egal şi de salt

Indiferent de clasa instrucţiunii, primii paşi de implementarea aunei instrucţiuni sunt la fel:

- ✓ se va trimite memoriei PC-ul şi se va extrage instrucţiunea din memorie
- ✓Se citesc 1 sau 2 registre - vom folosi câmpurile instrucţiunii pentru selectarea registrelor

Unităţile funcţionale şi legăturile dintre ele

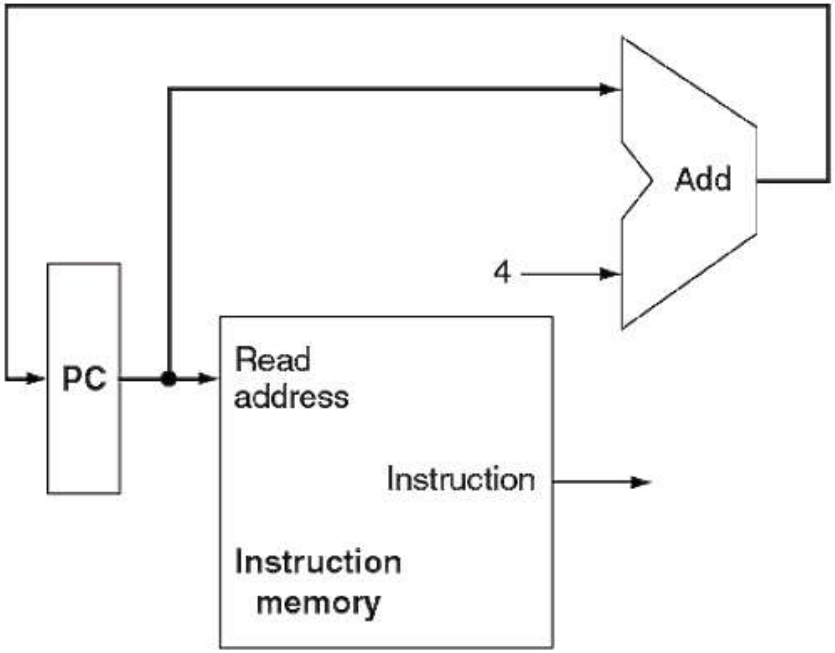


✓ Dacă instrucţiunea este de tip aritmetic sau logic => rezultatul din UAL trebuie scris într-un registru

✓ Dacă operaţia este de încărcare/memorare => rezultatul UAL va fi o adresă

✓ Pentru ramificaţii vom folosi ieşirea UAL în determinarea adresei următoarei Instrucţiuni de executat. Este necesar pentru aceasta să introducem o logică de control

Realizarea căii de date - determinarea instrucţiunii curente şi trecerea la următoarea instrucţiune

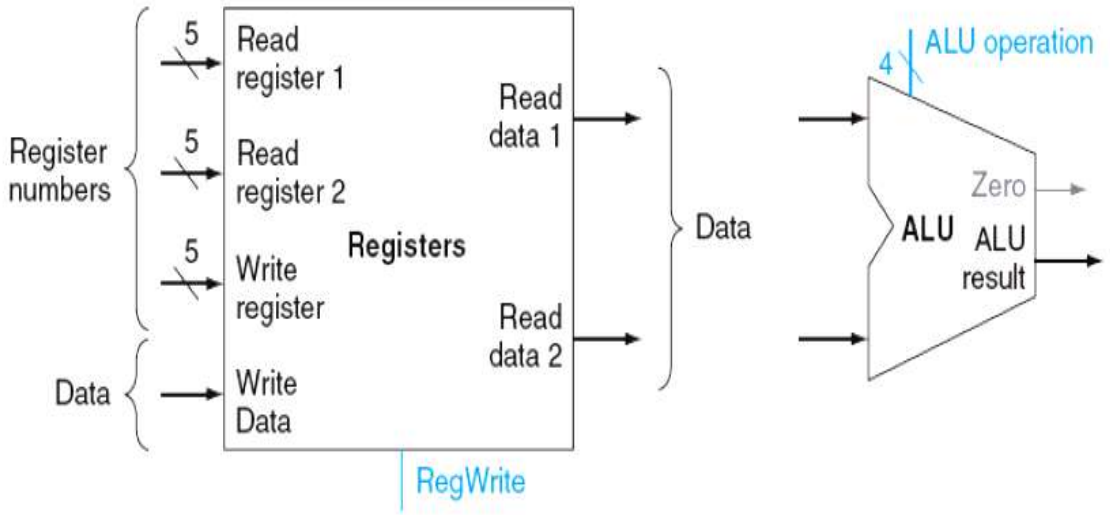


Extragem instrucţiunea din memorie

Incrementăm CP-ul cu 4 pentru Trecerea la instrucţiunea următoare

Instrucţiunea următoare este situată la o distanţă de 4 octeţi

Realizarea căii de date - instrucţiunile aritmetice şi logice



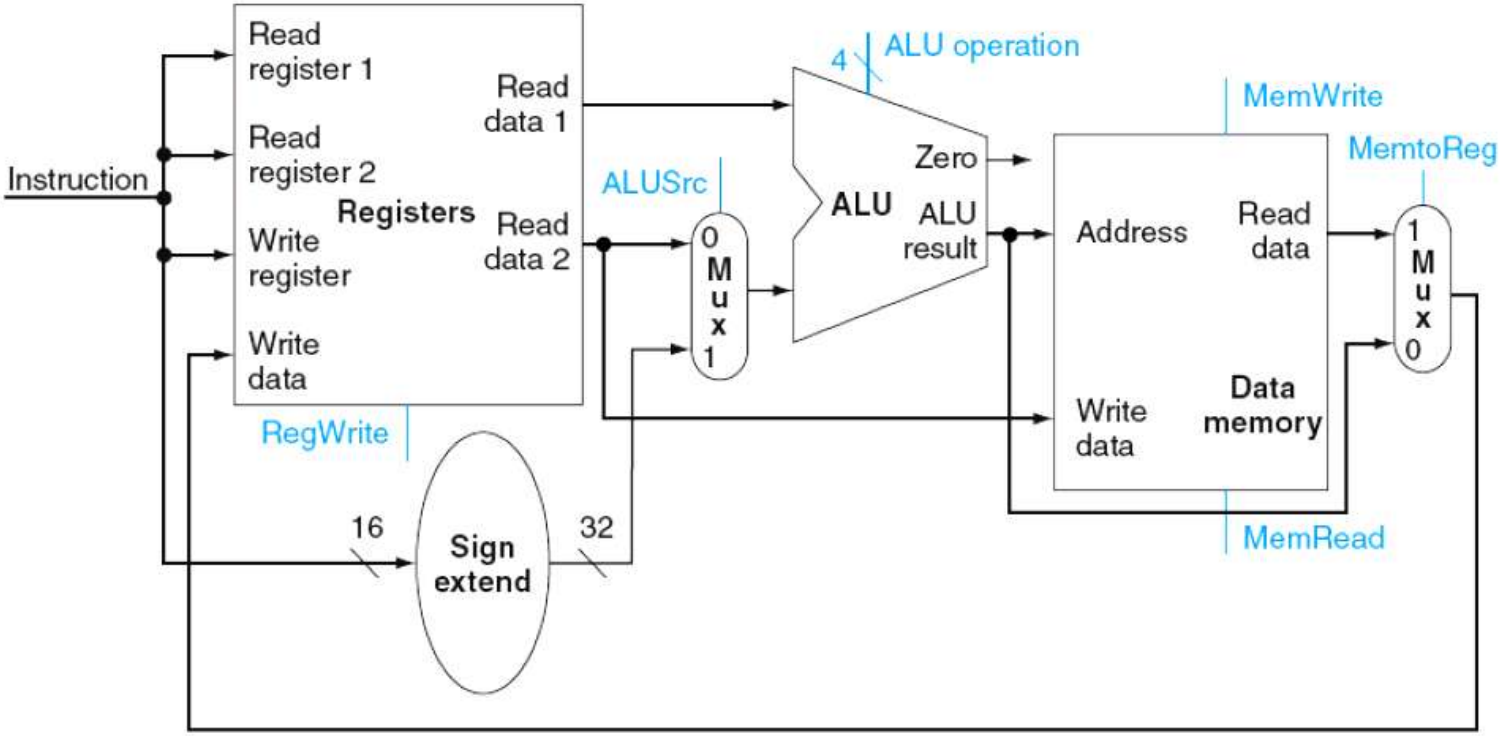
a. Registers b. ALU

Instrucţiunile de tipul R
add, sub, slt

Registre generale

UAL care operează cu valorile citite din registre

Realizarea căii de date - instrucţiunile de încărcare şi memorare



Instrucţiunile sunt:

lw $t1, valoare_deplasare($t2)
sw $t1, valoare_deplasare($t2)

Este necesară o unitate pentru a extinde semnul câmpului de deplasare din instr.

Realizarea căii de date - instrucţiunea beq

Instrucţiunea este formată din 2 registre care sunt comparate pentru egalitate şi o deplasare de 16 biţi

beq $t1, $t2, offset

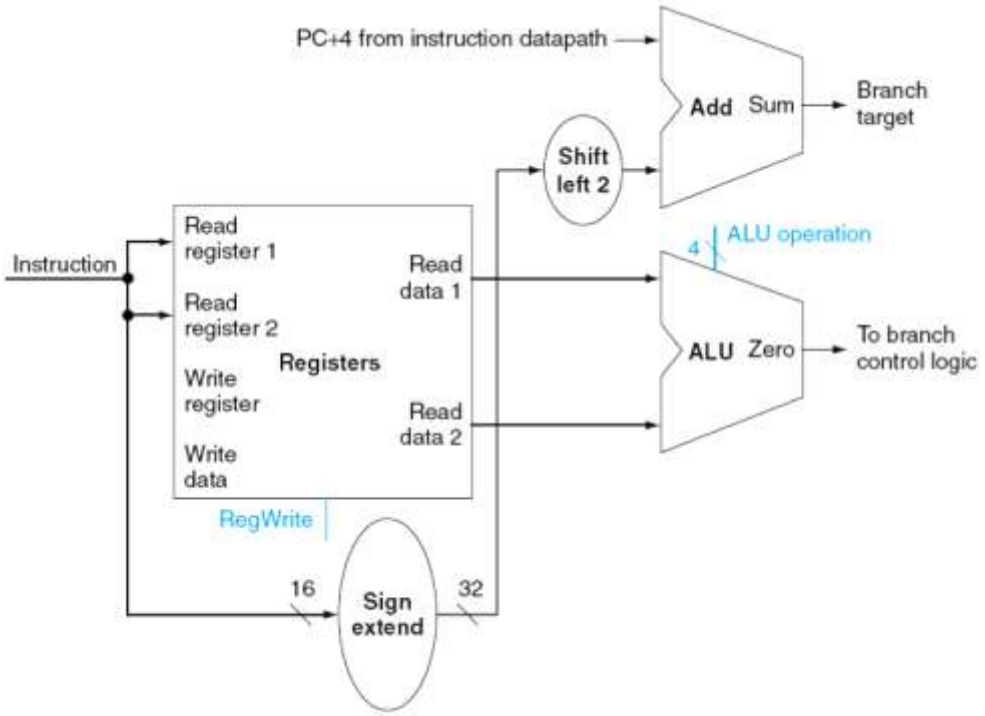
În vederea implementării acestei instrucţiuni trebuie determinată adresa obiectiv pentru ramificaţie => PC + câmpul de deplasare al instrucţiunii cu semnul extins

Particularităţi:

1. Baza pentru calculul adresei de ramificaţie este adresa instrucţiunii care urmează ramificaţiei => PC+4 valoarea ca bază

2. Câmpul deplasării din instrucţiune trebuie deplasat stânga cu 2 biţi deoarece deplasarea este la nivel de cuvânt => creşte domeniul efectiv al câmpului deplasării cu un factor de 4.

Realizarea căii de date - instrucţiunea beq



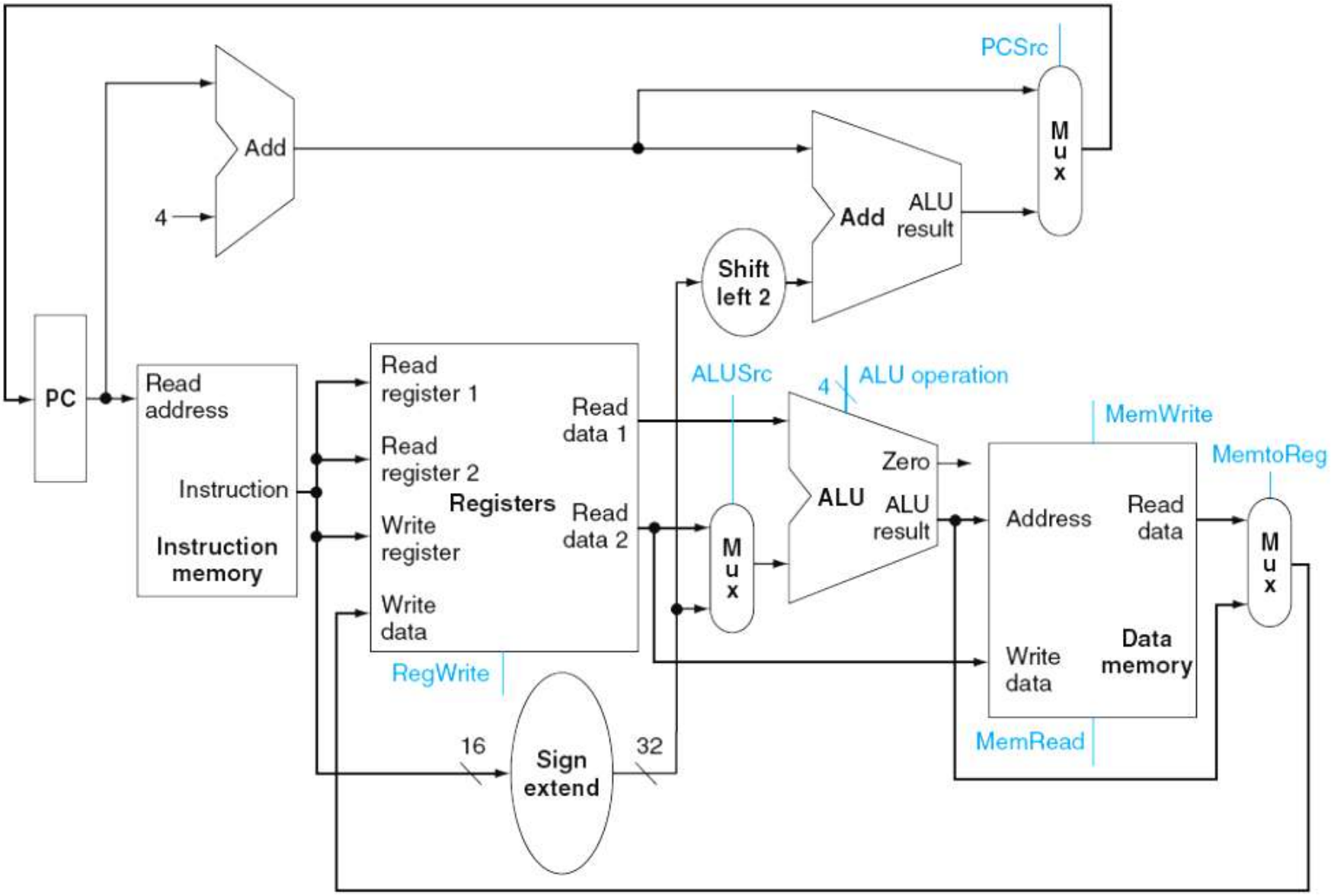
Operanzii sunt egali => adresa obiectiv pentru ramificaţie devine noul PC

Operanzii sunt diferiţi => PC-ul incrementat este noul PC

Trebuie realizate 2 operaţii:

1. Calcularea adresei obiectiv pentru ramificaţie
2. Compararea conţinutului registrelor

Obiectiv final - calea de date pentru execuţia într-un singur ciclu de ceas



Controlul UAL

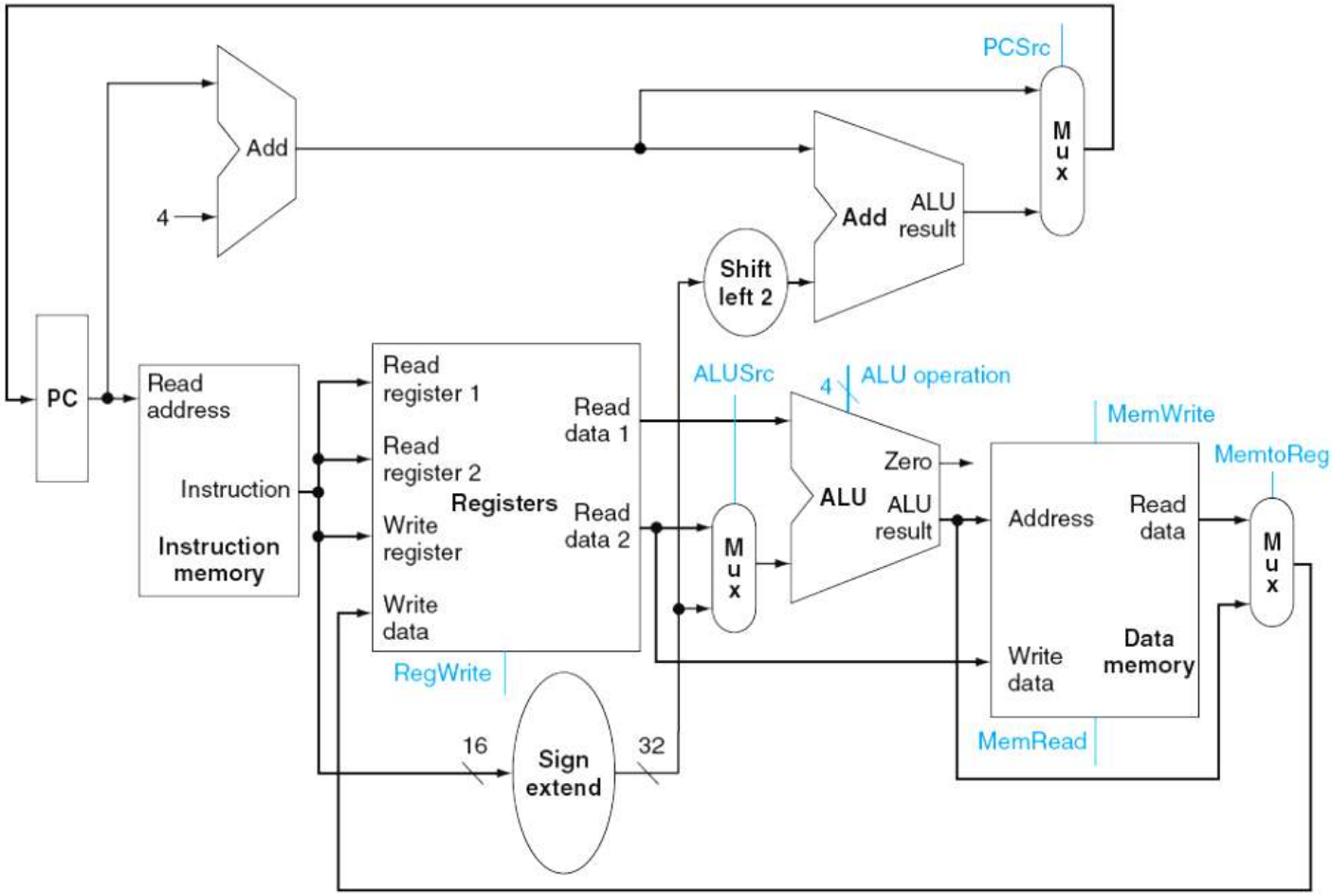
UAL are trei intrări de control din care folosite sunt 5

000	ŞI
001	SAU
010	+
110	-
111	setare la mai mic decât

Modalitatea de implementare - utilizăm 2 biţi (OpUAL) şi cei 6 biţi ai codului funcţiunii

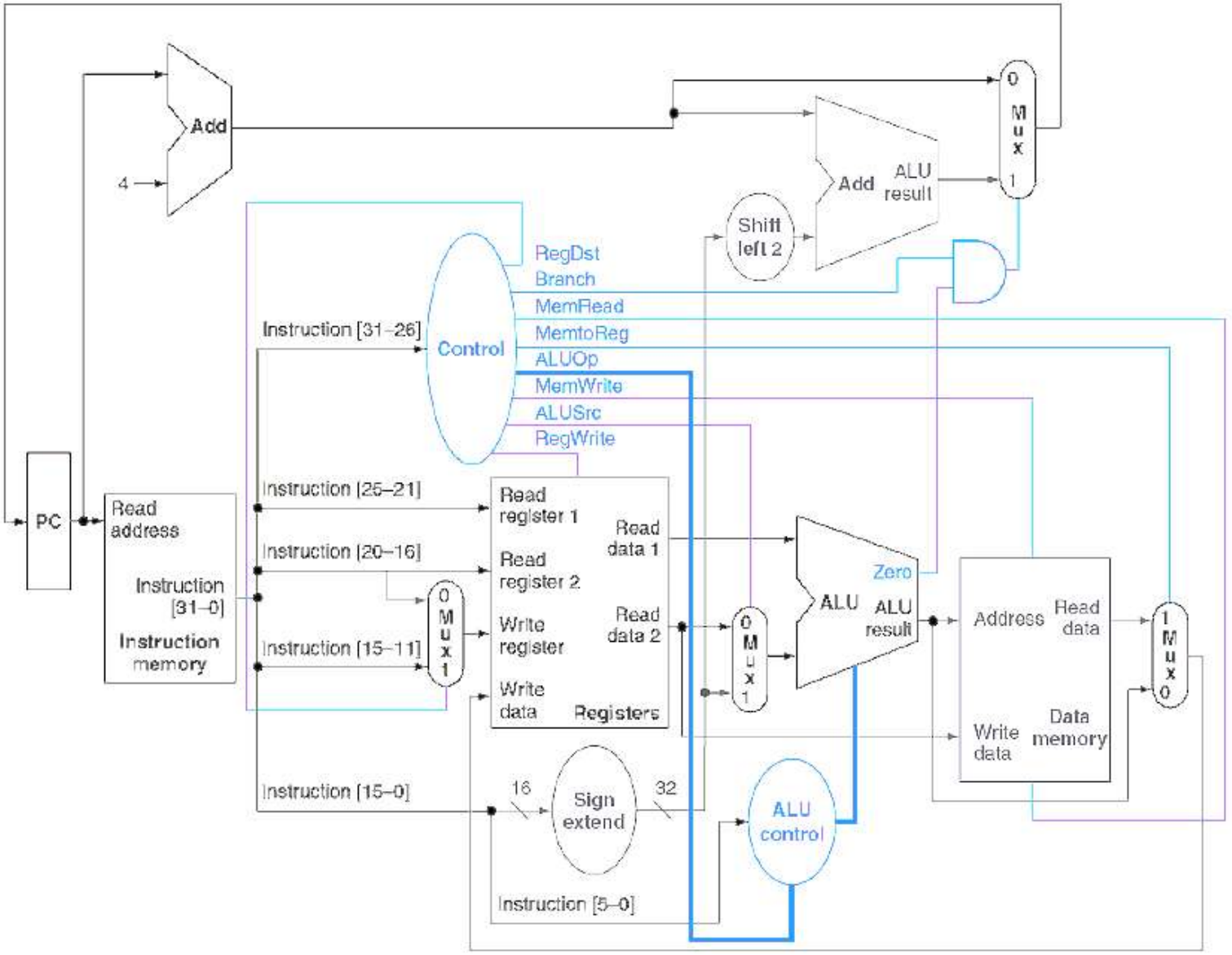
OpUAL	Operaţia	Câmpul funcţiunii	Acţiunea UAL	Intrare de control UAL
00	Încărcare cuv	xxxxxx	adunare	010
00	Memorare cuv	xxxxxx	adunare	010
01	Ramificaţie la egal	xxxxxx	scădere	110 etc

Schema controlului UAL completă

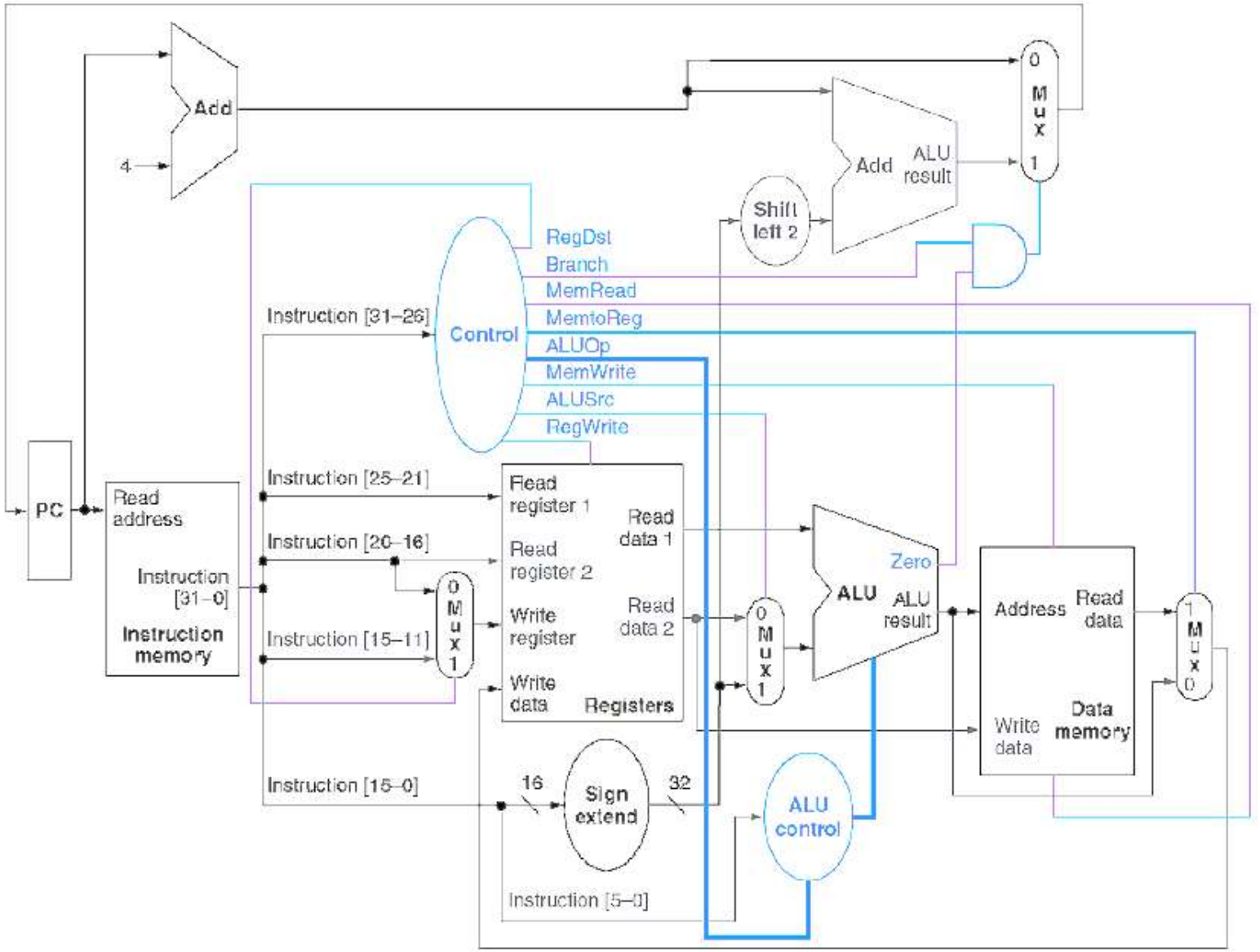


0 (31-26)	rs (25-21)	rt (20-16)	rd (15-11)	shamt(10-6)	funct(5-0)
35 sau 43 (31-26)	rs (25-21)	rt (20-16)	adresa(15-0)		
4 (31-26)	rs (25-21)	rt (20-16)	adresa(15-0)		

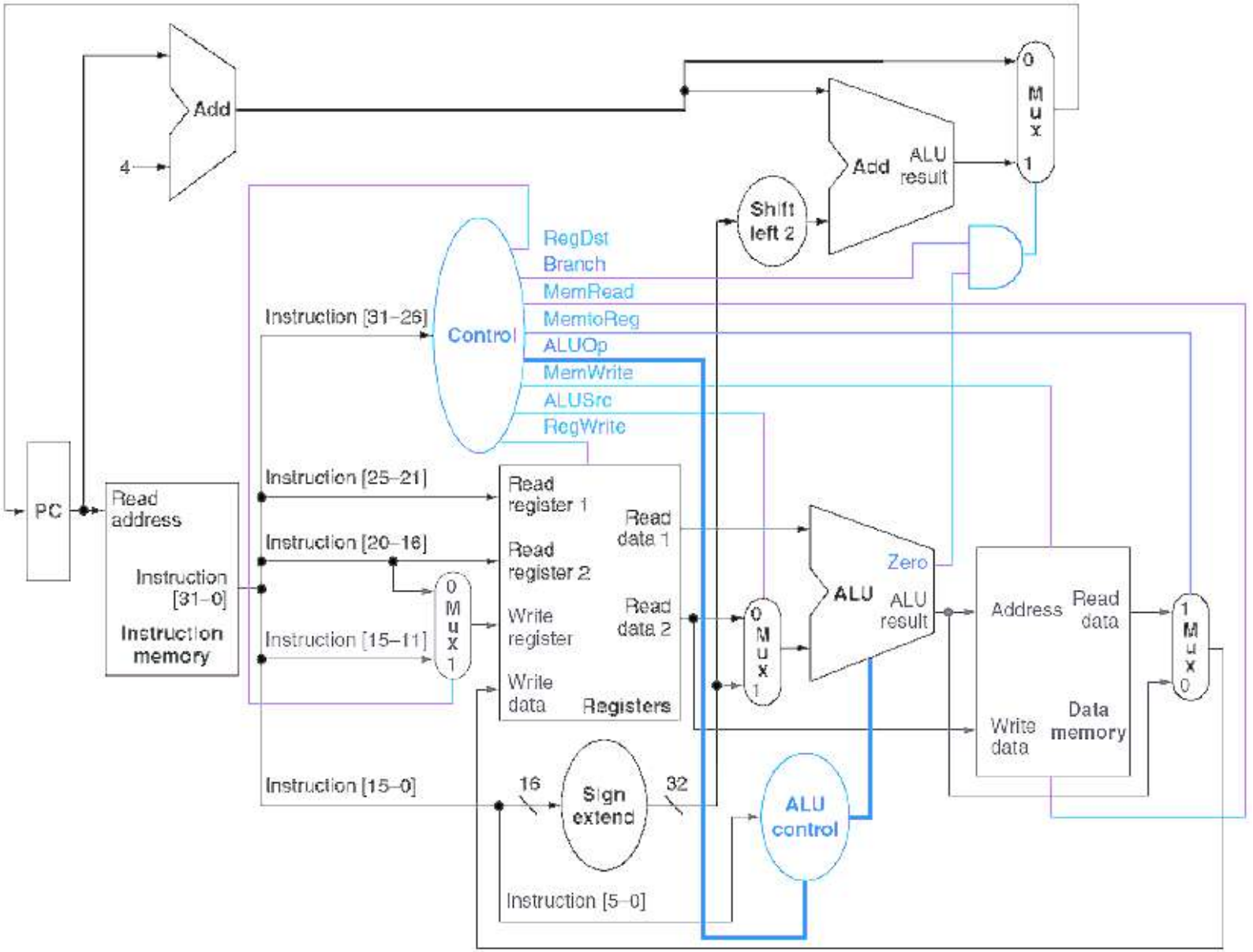
Execuţia instrucţiunii de tip R



Execuţia instrucţiunii de încărcare



Execuţia instrucţiunii de ramificare la egal

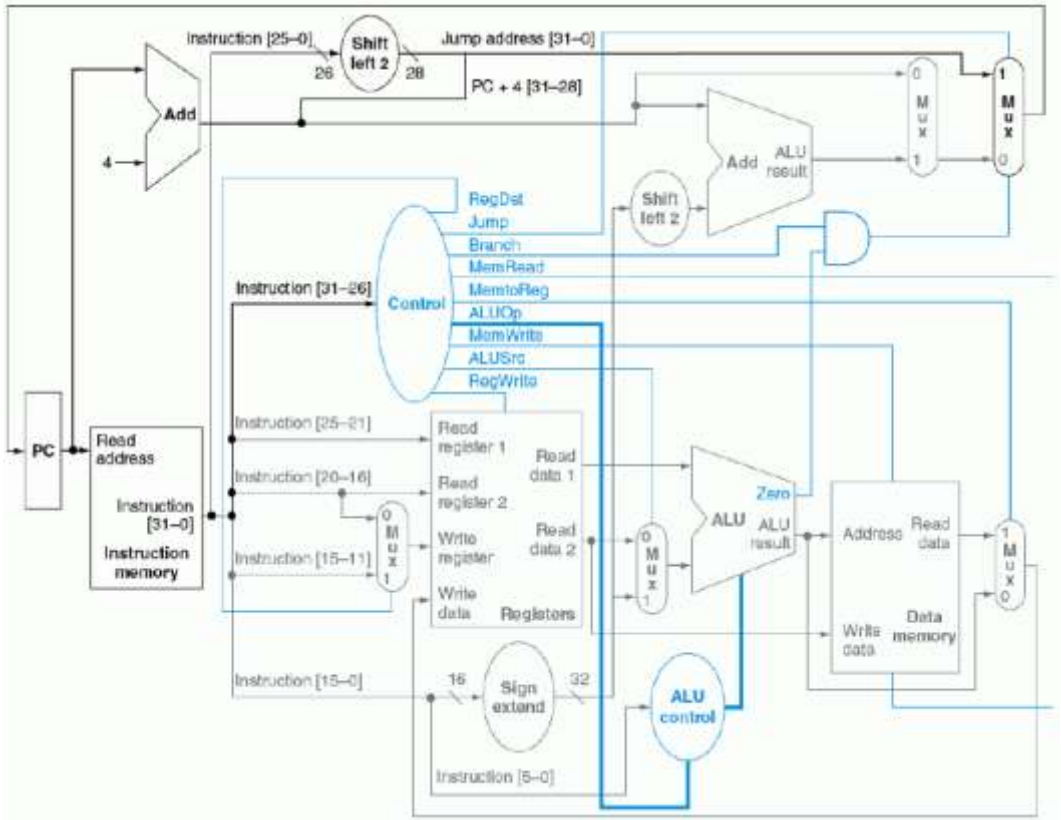


Implementarea instrucţiunii de salt

2 (31-26) adresa (25-0)

Calcularea PC-ului obiectiv

- ✓ cei 4 biţi MSB vin de la PC+4;
- ✓ câmpul imediat de 26 biţi ai instrucţiunii de salt;
- ✓ cei 2 biţi inferiori ai unei adrese de salt sunt întotdeauna 00



Folosim sau nu implementarea cu o singură perioadă de ceas ?

În cazul proiectării cu un singur ciclu de ceas, perioada ceasului trebuie să aibă aceeaşi durată pentru fiecare instrucţiune => ciclul de ceas este determinat de cea mai lungă cale posibilă prin maşină

Cazul cel mai frecvent nu are o execuţie rapidă

Fiecare unitate funcţională poate fi utilizată o singură dată într-un ciclu de ceas => repetarea un ităţilor funcţionale => creşterea preţului implementării

Metode alternative :

implementarea cu mai multe cicluri

PIPELINE

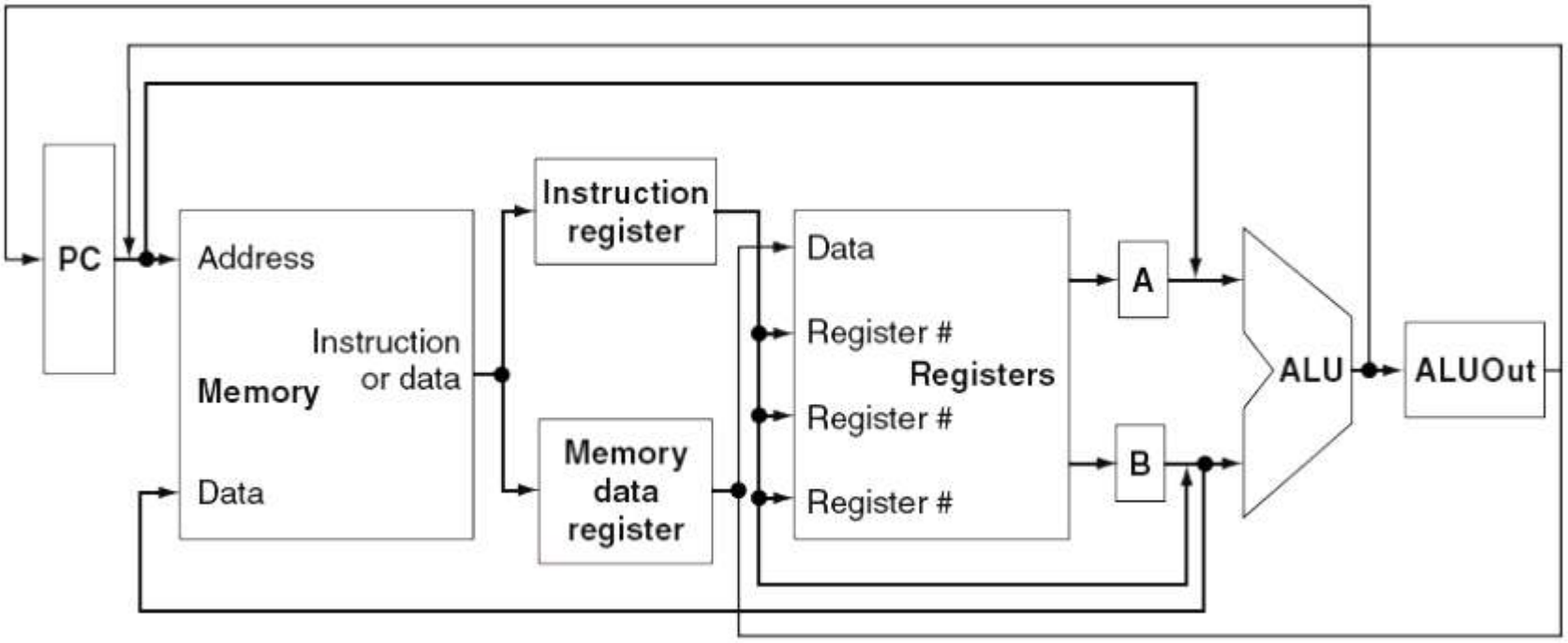
Metodă INEFICIENTĂ, neutilizată în practică

IMPLEMENTAREA CU MAI MULTE CICLURI

Într-o implementare cu mai multe cicluri, fiecare pas al execuţiei va necesita o perioadă de ceas.

Unitatea funcţională este utilizată mai mult decât o singură dată pe instrucţiune – utilizată evident în cicluri de ceas diferite

Acestea sunt avantajele majore ale implementării cu mai multe cicluri de ceas



La finalul unui ciclu de ceas toate datele care sunt folosite în ciclurile următoare trebuie memorate în elemente de stare

Datele folosite de instrucțiunile următoare într-un ciclu ulterior de ceas vor fi memorate în elementele de stare vizibile programatorului.

Datele folosite de aceeași instrucțiune într-un ciclu ulterior de ceas trebuie memorate în registrele suplimentare

SE PRESUPUNE

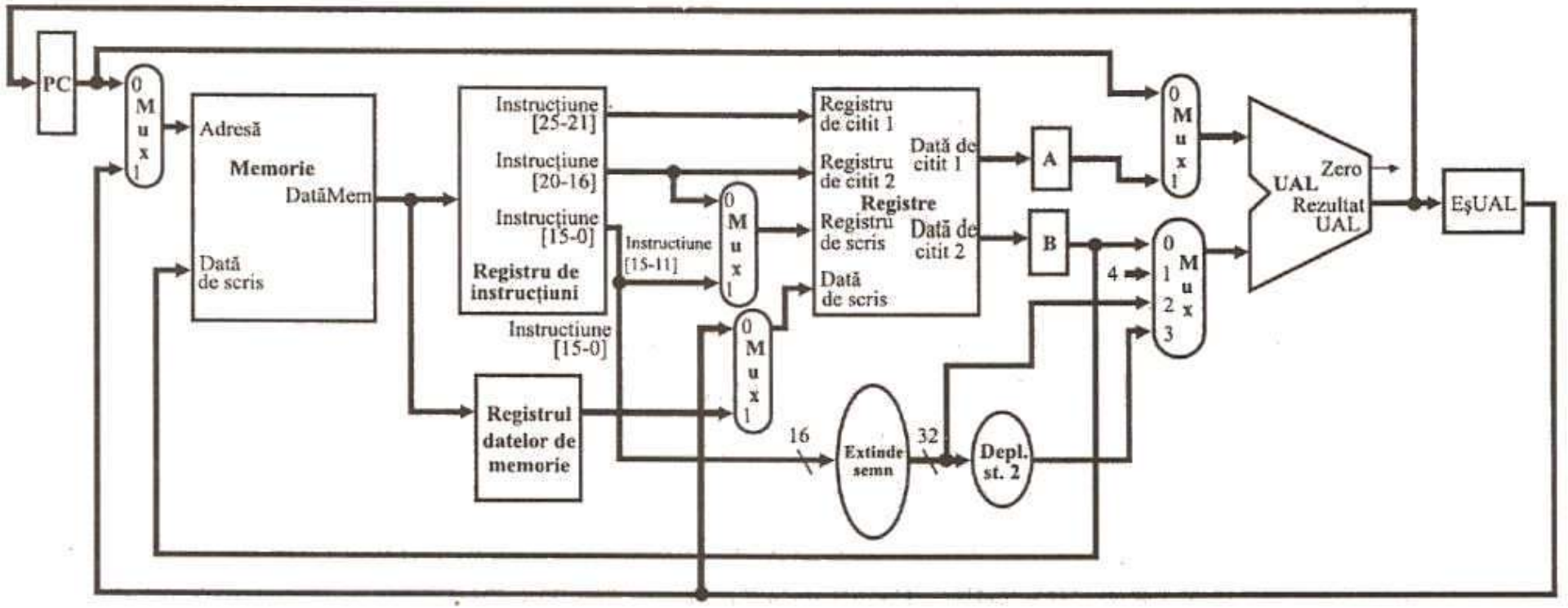
Durata ciclului de ceaspoate deservi cel mult:

- un acces la memorie
- un acces la fișierul de registre – 2 citiri și o scriere
- o operație UAL

Registrul de instructiuni (RI) și Registrul datelor de memorie (MDR)

Registrele A și B

Registrul EșUAL



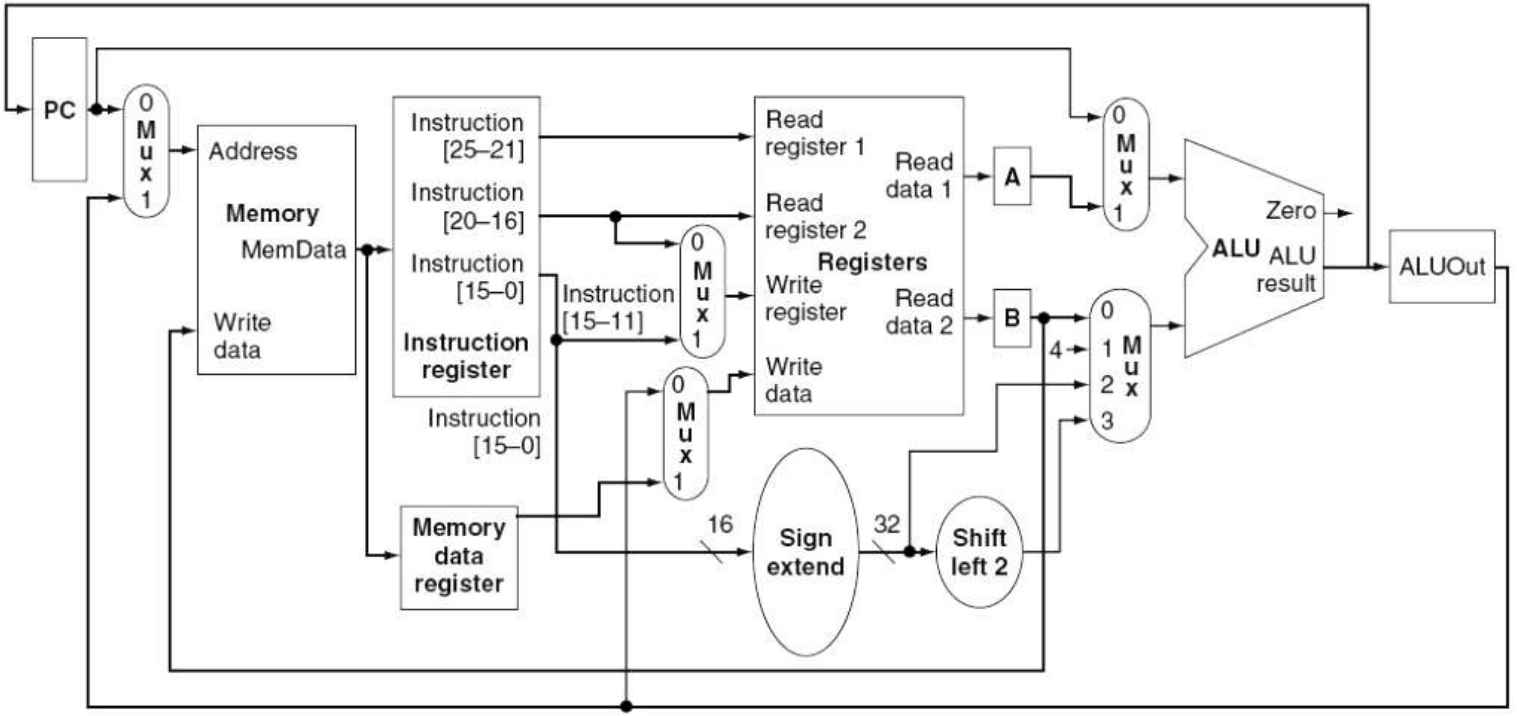
Folosim mai multe unităţi funcţionale în comun, deci trebuie adăugate multiplexoare

Exp: Avem o singură memorie atât pentru date cât şi pentru instrucţiuni – PC şi OpUAL

Folosim un singur UAL în loc de 3 ca în cazul implementării cu un singur ciclu de ceas, deci vom avea următoarele schimbări

• un multiplexor suplimentar adăugat primei intrări în UAL

• multiplexorul celei de-a doua intrări în UAL din MUX2 se transformă în MUX4



CE AM OBŢINUT ?

- Reducerea numărului de un itați de memorie de la 2 la 1
- Eliminarea a două sumatoare

AM OBŢINUT O REDUCERE SEMNIFICATIVĂ A COSTULUI HARDWARE-ULUI

CE NU AM IMPLEMENTAT ?

- ramificaţiile
- salturile

Având în vedere salturile precum şi ramificaţiile există 3 surse posibile pentru PC

- ieşirea UAL – PC + 4

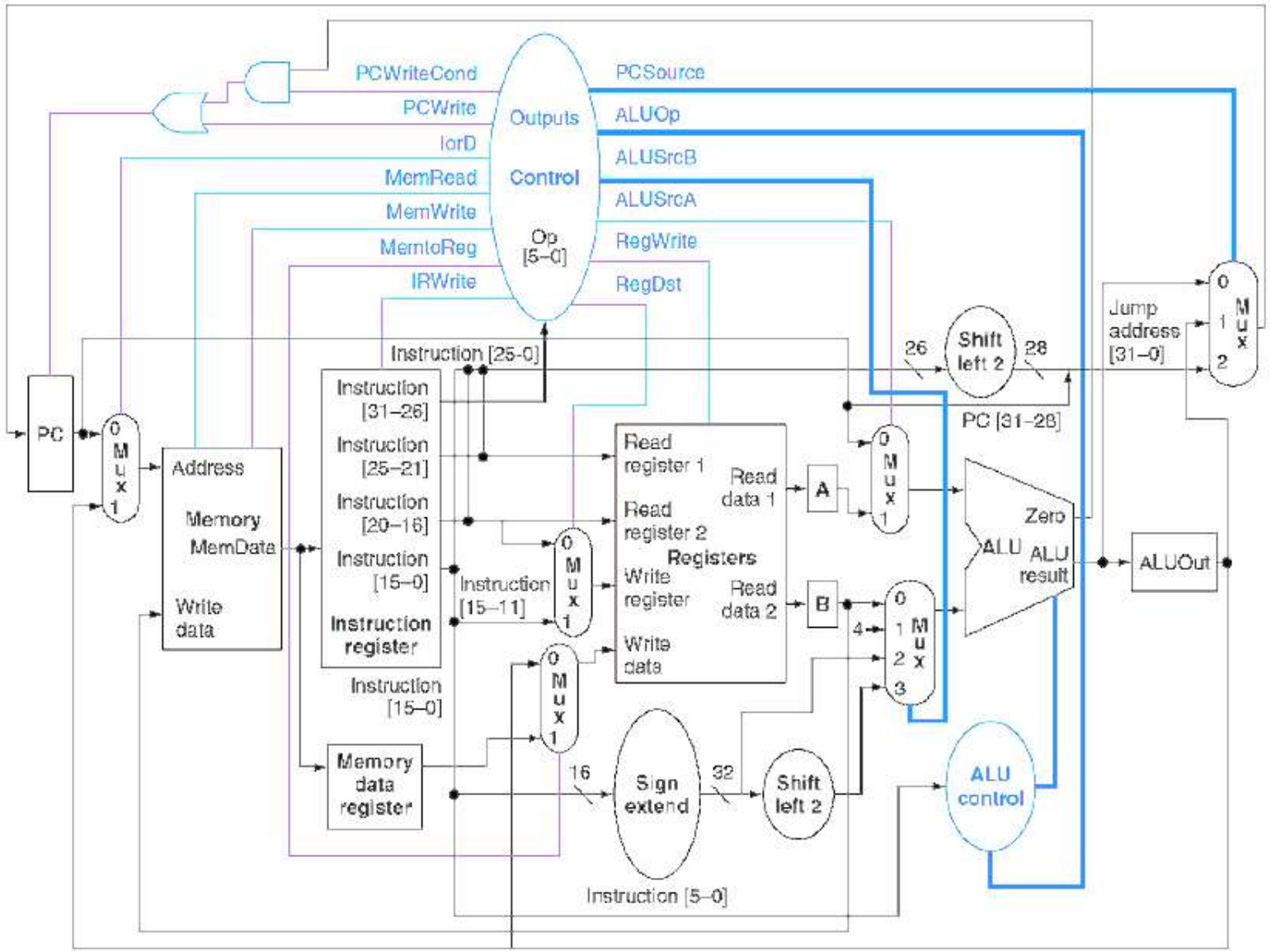
- registrul EşUAL cel care memorează adresa obiectiv pentru ramificaţie evident după determinarea sa

- cei 26 de biţi inferiori ai lui IR deplasaţi spre stânga cu 2 poziţii şi concatenaţi cu cei 4 biţi superiori ai PC-ului incrementat – sursa pentru instrucţiunea de salt

CONCLUZIE – PC-ul va trebui scris atât condiţionat cât şi necondiţionat !!!

Vom folosi magistrale partajate

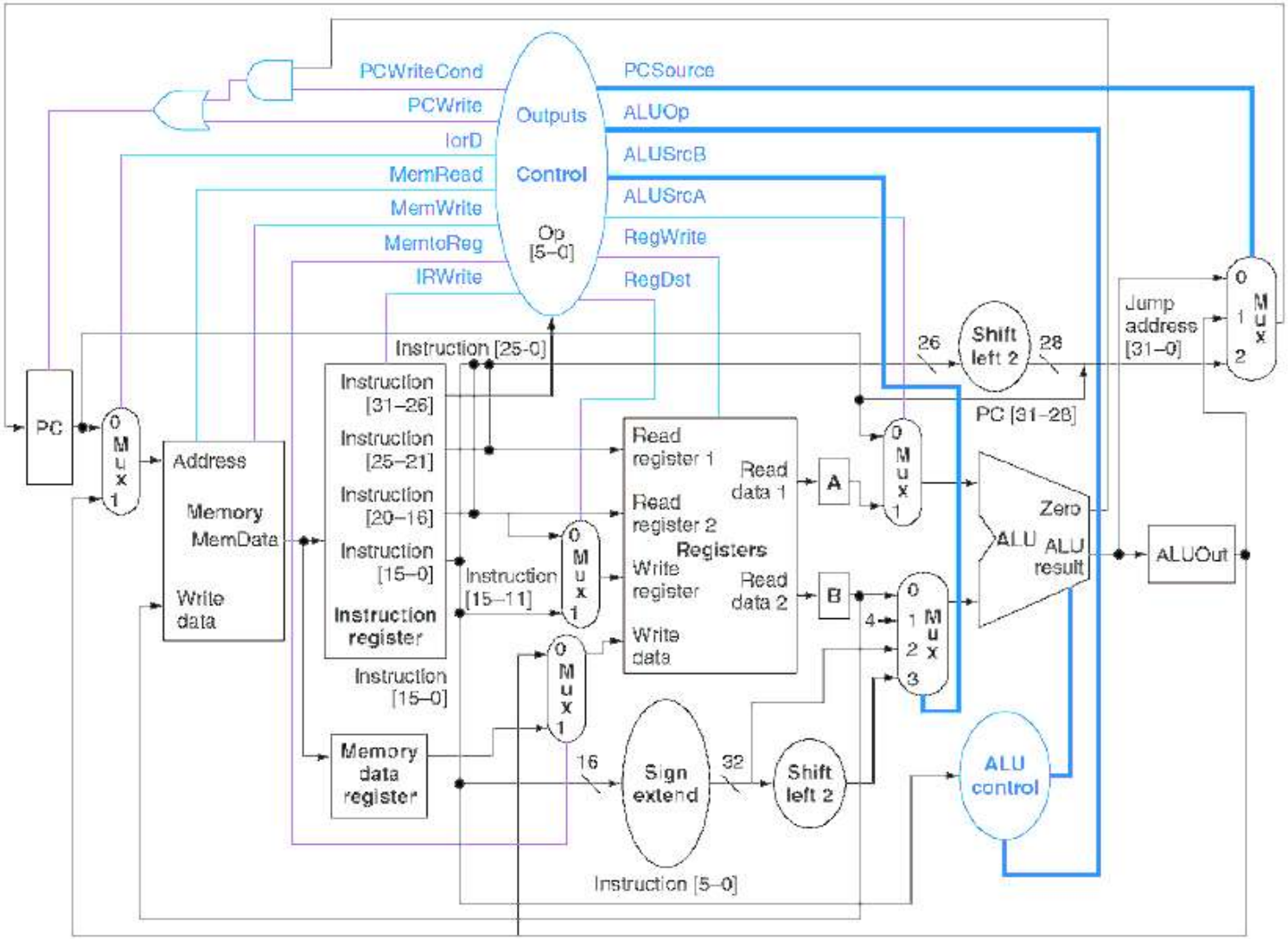
Calea de date completă cu liniile de control necesare



Paşii de execuţie a instrucţiunii

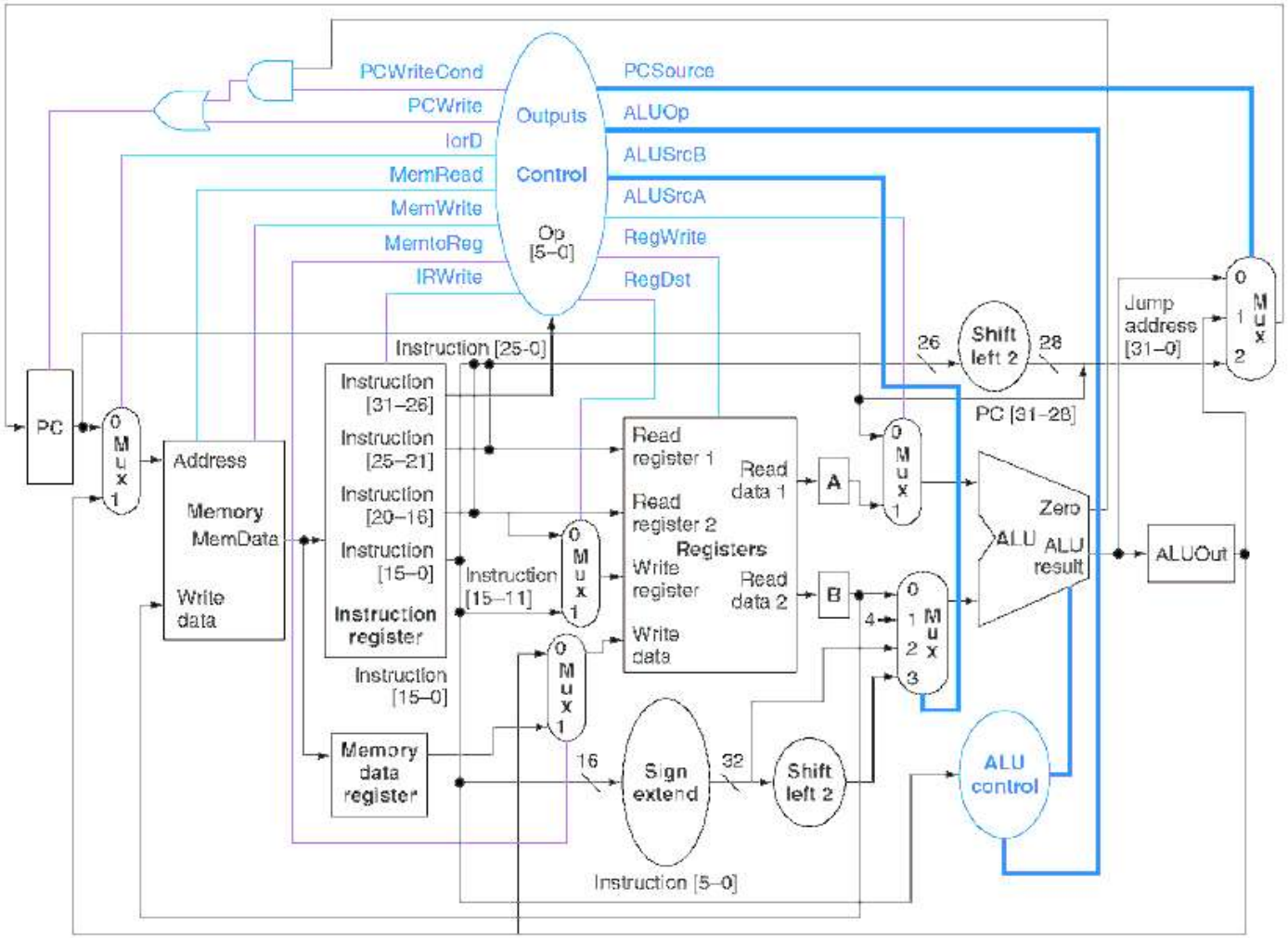
EXTRAGEREA INSTRUCŢIUNII

IR = Memorie(PC);
PC = PC+4



DECODIFICAREA INSTRUCȚIUNII ŞI EXTRAGEREA REGISTRELOR

A = Reg(IR(25-21))
B = Reg (IR(20-16))
EşUAL = PC + (semn-extins(IR(15-0)) << 2);



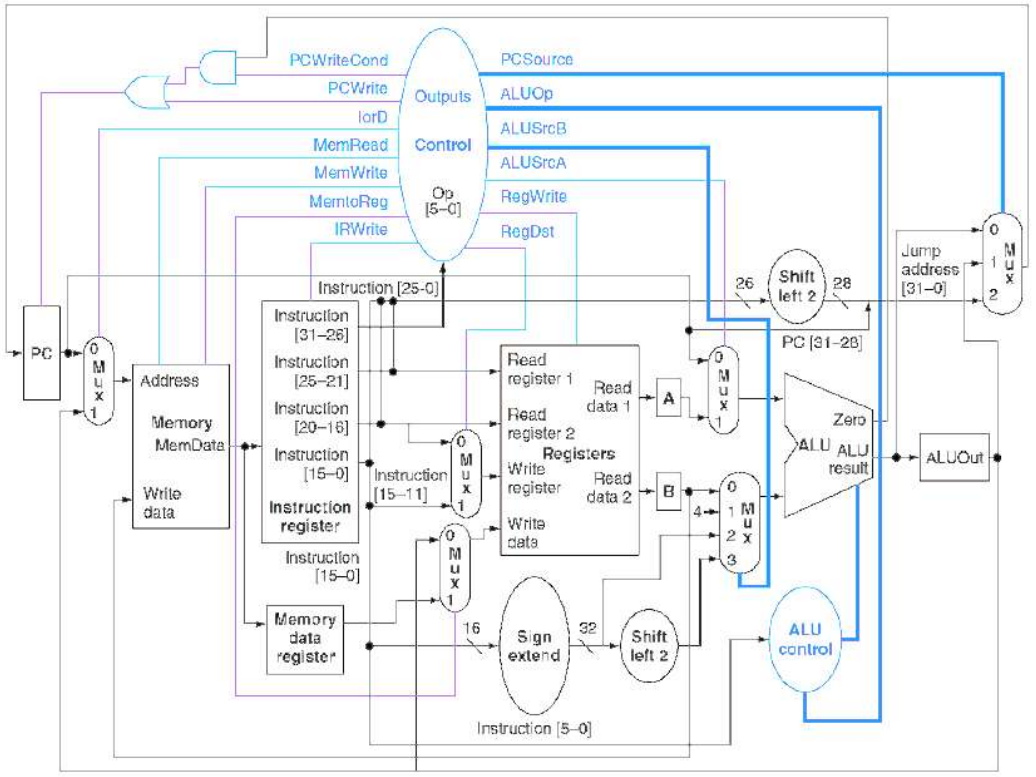
Execuţia, calculul adresei de memorie sau terminarea ramificaţiei

Referinţă la memorie – IeşireUAL = A + semn-extins (IR(15-0))

Instrucţiune tip R – IeşireUAL = A op B

Ramificaţie – dacă (A==B) atunci PC=EşUAL

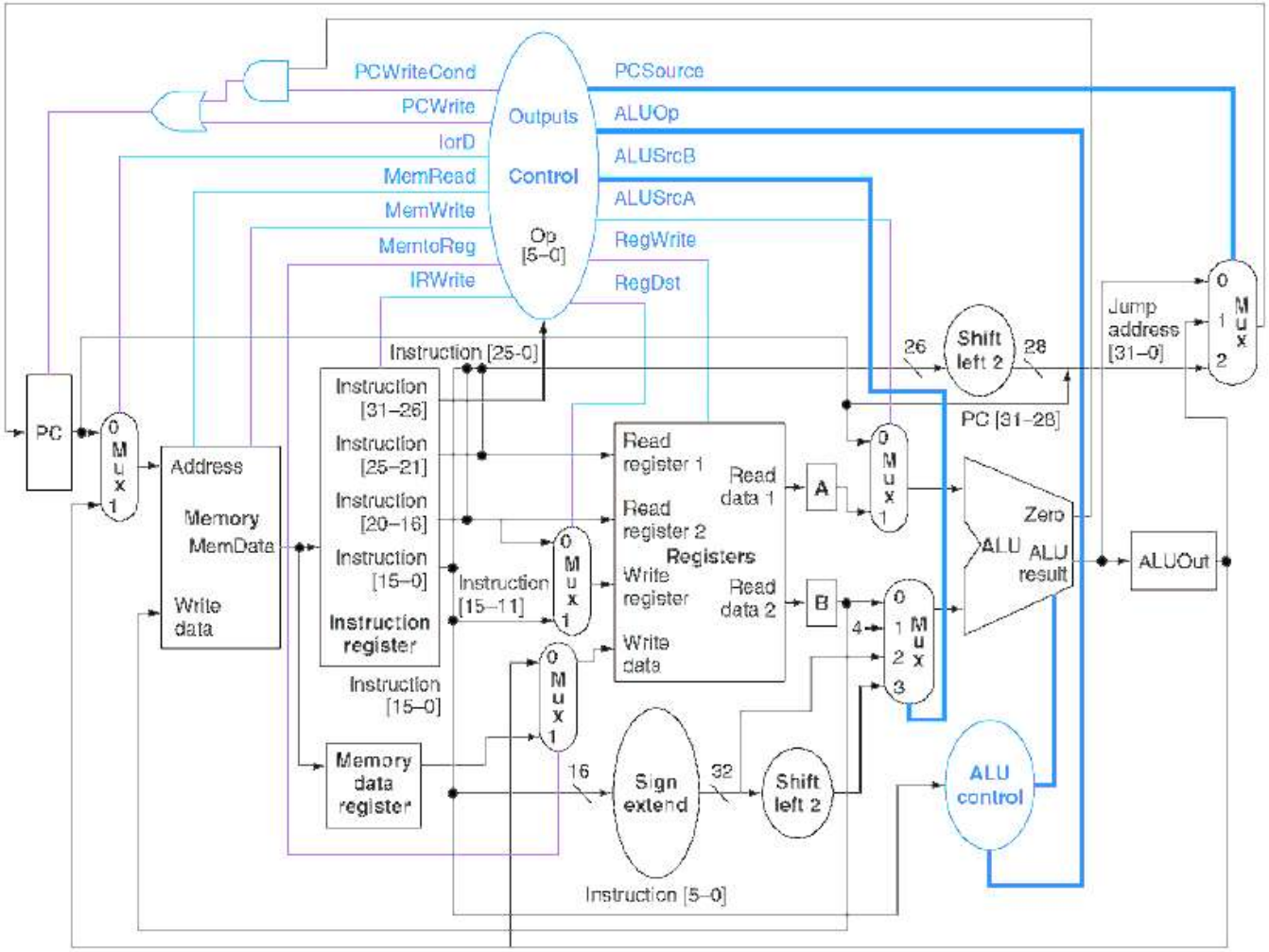
Salt – PC = PC(31-28) || IR(25-0) << 2



Pasul de acces la memorie sau de terminare a instrucţiunii de tip R

O instrucţiune de încărcare/memorare accesează memoria, iar o instrucţiune aritmetică-logică îşi scrie rezultatul.

Valoarea citită din memorie este scrisă în MDR de unde va fi folosită în ciclul următor

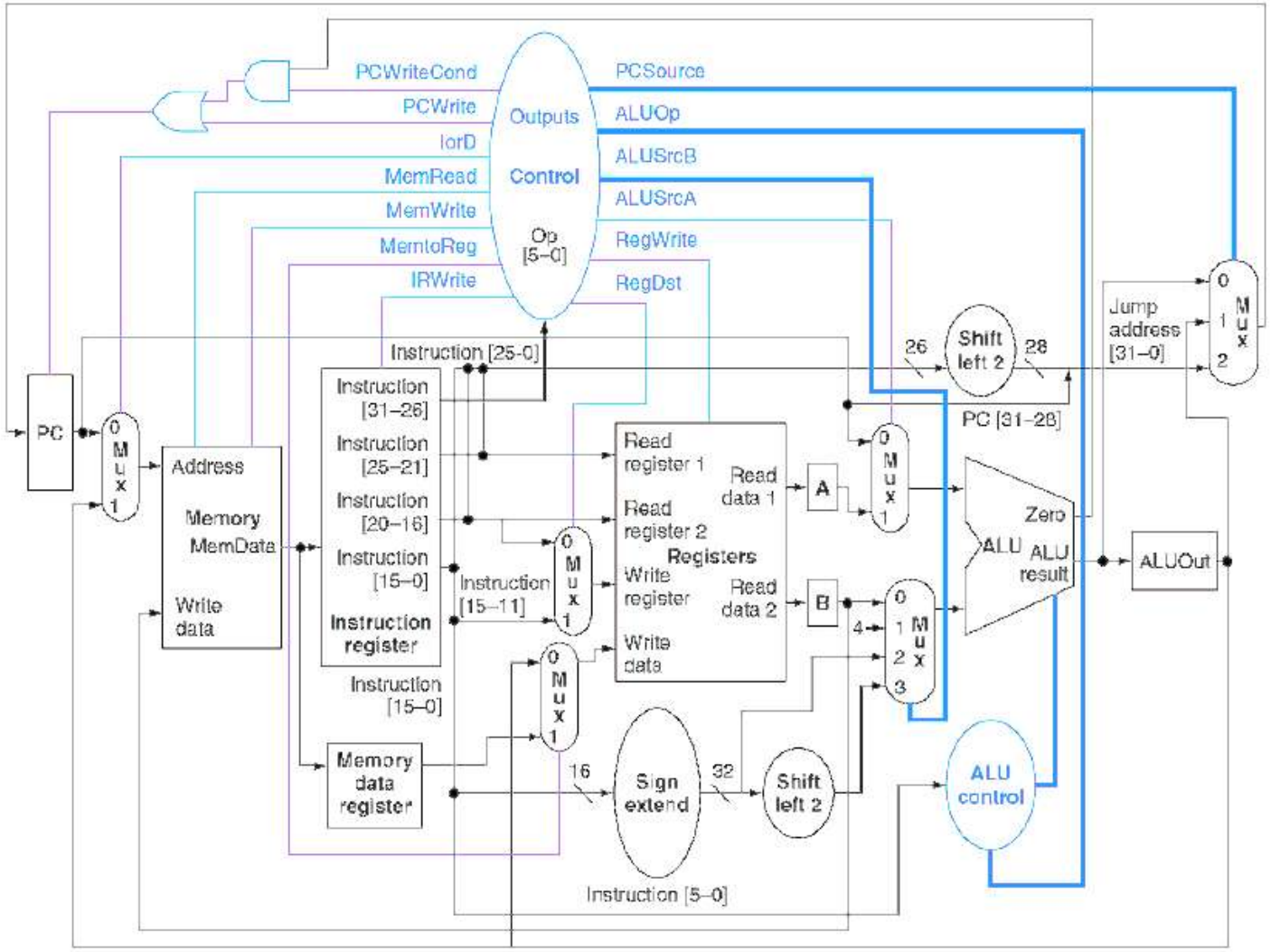


MDR = Memorie (EşUAL)

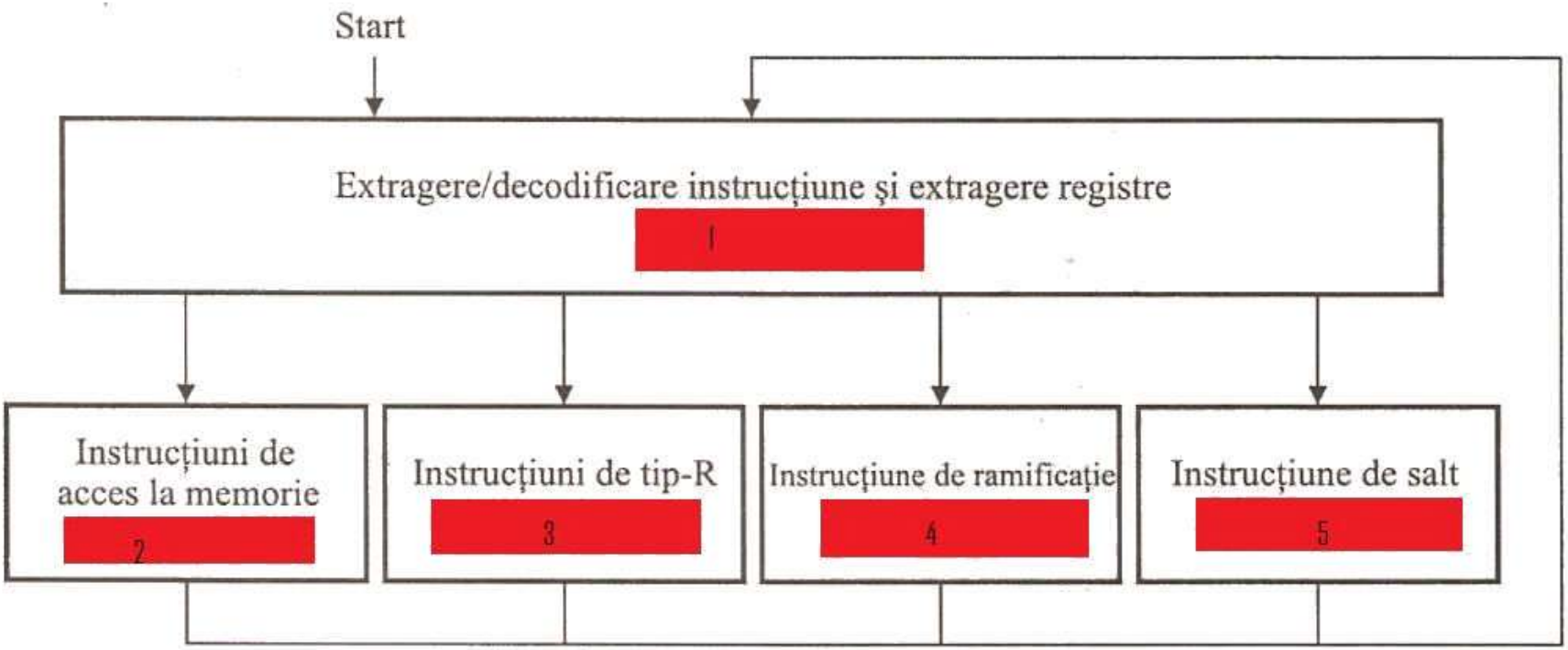
Reg(IR(15 -11)) = EşUAL

Pasul de terminare a citirii memoriei

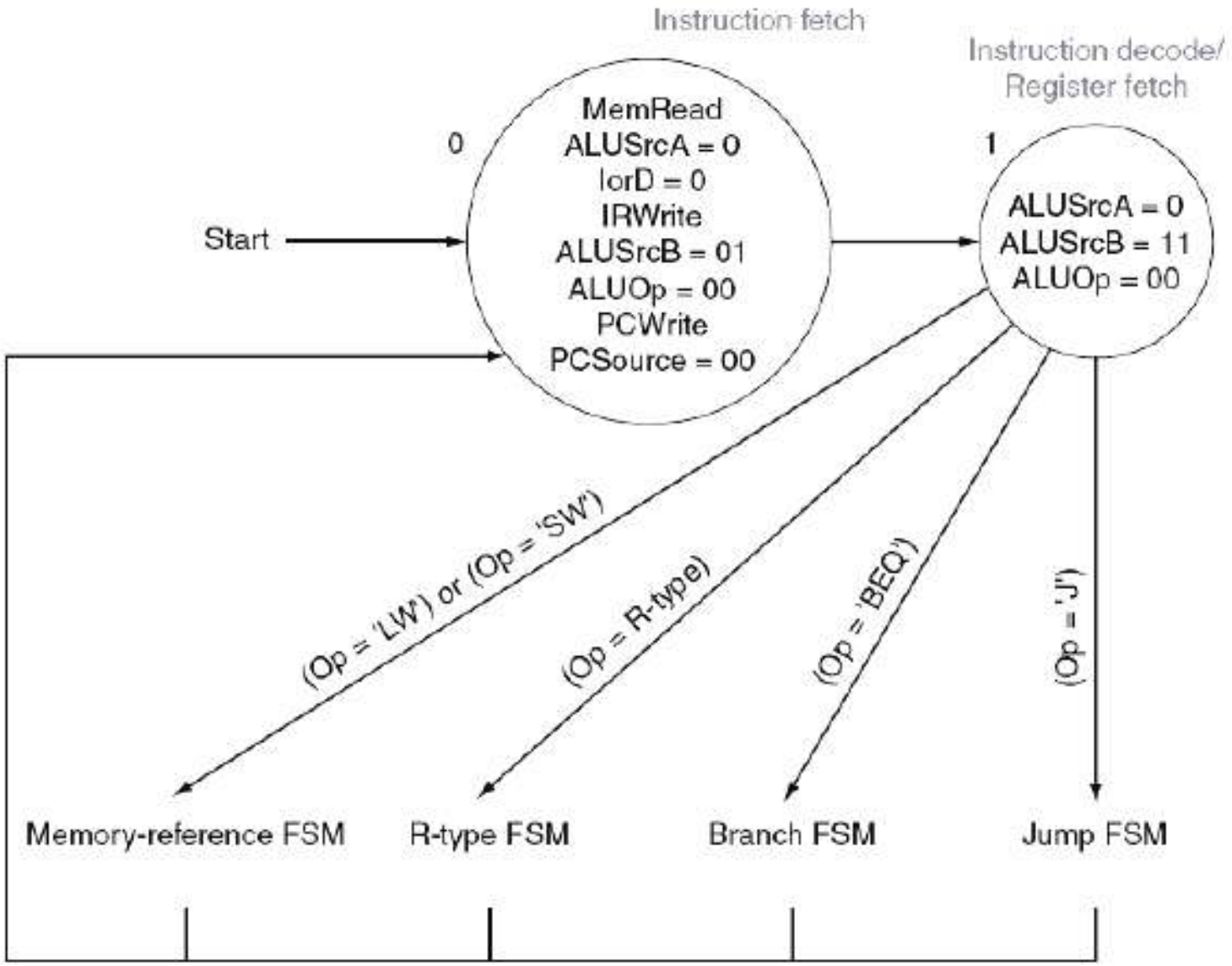
Reg(IR(20-16)) = MDR



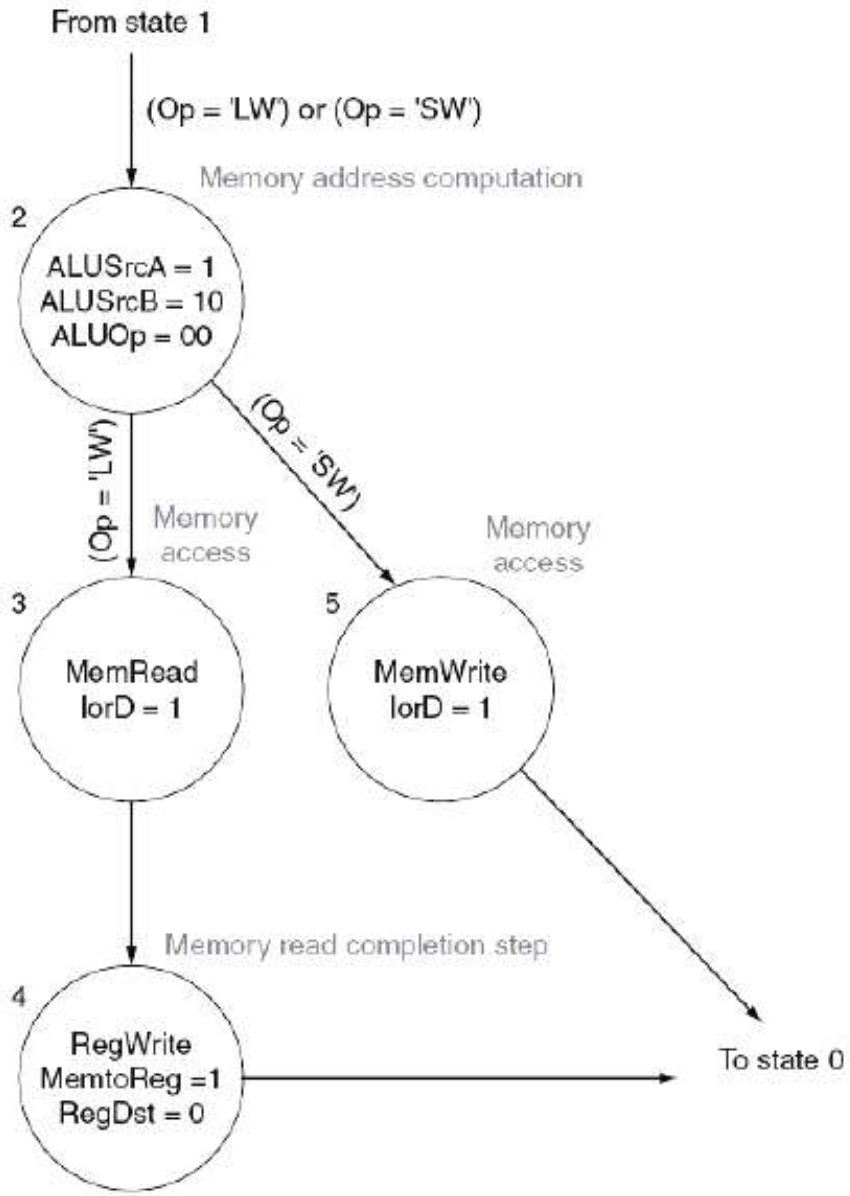
Definirea controlului



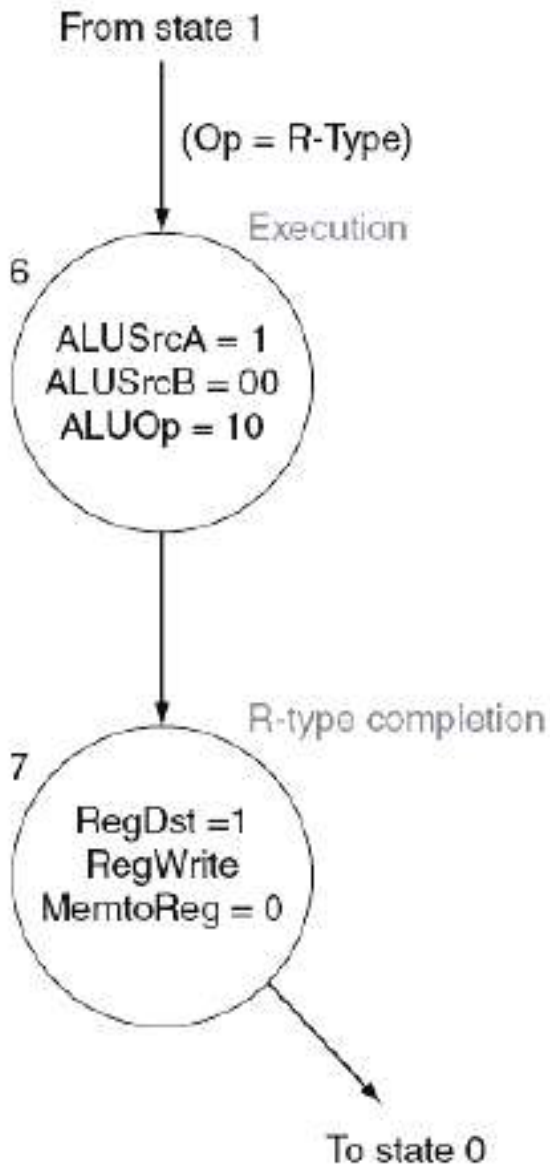
Instrucţiunea este extrasă şi decodificată



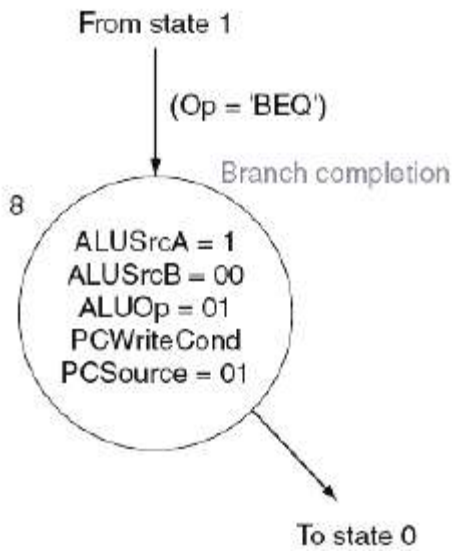
Controlul instrucţiunilor de referire a memoriei

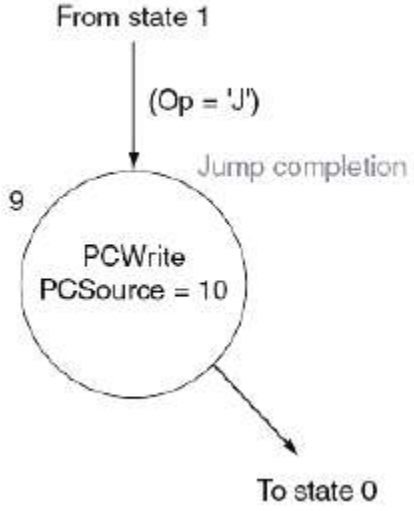


Instrucţiunile de tip R



Instrucţiunile de salt şi ramificaţie şi de salt





MICROPROGRAMAREA

Microprogramarea este folosită pentru reducerea complexității proiectării controlului

Fiecare microinstrucțiune defineşte setul de semnale de control ale căii de date care trebuie activate într-o singură stare dată

Foarte importantă este succesiunea microinstrucțiunilor

Microinstrucțiunea poate fi privită ca o secvență de câmpuri ale căror funcțiuni sunt corelate.

Cum definim formatul microinstrucțiunii ?

Trebuie stabilite numărul de câmpuri dintr-o microinstrucțiune precum şi semnalele de control afectate de fiecare câmp.

1. Formatul microinstrucțiunii trebuie ales astfel încât să se simplifice reprezentarea.

2. Trebuie ca scrierea microinstrucțiunilor consistente să fie imposibilă

Semnalele care nu sunt activate simultan niciodată pot folosi în comun acelaşi câmp

Exemplu de împărțire pe câmpuri a unei microinstrucțiuni:

ControlUAL	- specifică operația efectuată de UAL
SRC1	- sursă operand 1
SRC2	- sursă operand 2
CtrlReg	- scrire/citire a fişierului de registre şi sursa valorii pt scriere
Memorie	- scrierea/citirea şi sursa pentru memorie
CtrlScriePC	- specifică scrierea PC-ului
Secvență	- modalitatea de alegere a următoarei instr.

ALEGEREA INSTRUCȚIUNII URMĂTOARE

1. Incrementarea microinstrucțiunii curente – comportare secvențială (mod implicit) – în câpul secveță punem eticheta SEQ

2. Se transferă controlul la microinstrucțiunea care începe execuția următoarei instrucțiuni – ciclu FETCH – în câpul secvență punem FETCH

3. Se alege următoarea microinstrucțiune pe baza intrării în unitatea de control – distribuție

Câmp	Valori pt câmp	Funcție
Etichetă	Orice şir	Folosit pentru specificarea etichetelor în controlul secvențelor de microcod. Etichetele terminate în 1 sau 2 sunt folosite pentru distribuție cu tabel de salt, indexat cu cod operație
Control UAL	ADD	UAL adună
	Subst	UAL scade – implementeză comparația
	Func code	Foloseşte câmpul ***funct***

Câmp	Valori pt câmp	Funcție
SRC 1	PC	PC intrare pentru UAL
	A	A folosit ca primă intrare în UAL
SRC 2	B	B a doua intrare în UAL
	4	Constanta 4 ca a doua intrare în UAL
	Extend	Ieşirea unității de extindere semn ca a doua intrare în UAL
	Extshft	Foloseşte ieşirea unității deplasare cu 2 ca a doua intrare în UAL
Ctrl registru	Read	Citeşte 2 registre folosind pentru identificarea lor câpurile rs şi rt
	Write ALU	Scrie fişierul de registre folosind pt numărul registrului câmpul rt din IR, iar pentru dată conținutul EşUAL
	Write MDR	Scrie fişierul de registre folosind pentru numărul registrului câmpul rt din IR iar pentru dată conținutul MDR
Memorie	Read PC	Citeşte memoria având ca adresă PC-ul; scrie rezultatul în IR şi MDR
	Read ALU	Citeşte memoria având ca adresă EşUAL; scrie rezultat în MDR
	Write ALU	Scrie memoria având ca adresă EşUAL şi conținutul lui B ca dată

Câmp	Valori pt câmp	Funcție
Ctrl ScriePC	ALU	Scrie conținutul lui UAL în PC
	ALUOut-cond	Dacă ieşirea UAL-Zero este activă, scrie în PC conținutul registrului EşUAL
	Jump address	Scrie în PC adresa de salt a instrucțiunii
Secvență	seq	Alege secvența următoare în microinstrucțiune
	Fetch	Merge la prima microinstrucțiune pentru a începe o instrucțiune nouă
	Dispatch i	Distribuie adresa folosind ROM-ul specificat de i (1 sau 2)

Crearea microprogramului

Se vor eticheta instrucțiunile din microprogram cu etichete simbolice care pot fi folosite pentru specificarea conținutului tablourilor de distribuție.

Paşii 1 şi 2 din execuția unei instrucțiuni:

Extragerea instrucțiunilor

Decodificarea instrucțiunilor şi calcularea PC-ului secvențial cât şi a PC-ului obiectiv pentru ramificație

Etch	Ctrl UAL	SRC1	SRC2	Ctrl Reg	Mem	Ctrl ScriePC	Secv
Fetch	Add	PC	4		Read PC	UAL	Seq
	Add	PC	Extshft	Read			Dispatch 1

Microprogramul pentru instrucțiunile de referire a memoriei

Etch	Ctrl UAL	SRC1	SRC2	Ctrl Reg	Mem	Ctrl ScriePC	Secv
mem1	Add	A	Extend				Dispatch 2
lw2					Read ALU		Seq
				Write MDR			Fetch
sw2					Write ALU		fetch

Microinstrucțiunile pentru instrucțiunile de tip R

Etch	Ctrl UAL	SRC1	SRC2	Ctrl Reg	Mem	Ctrl ScriePC	Secv
Rformat 1	Func code	A	B				Seq
				Write ALU			fetch

Instrucțiunea de ramificație

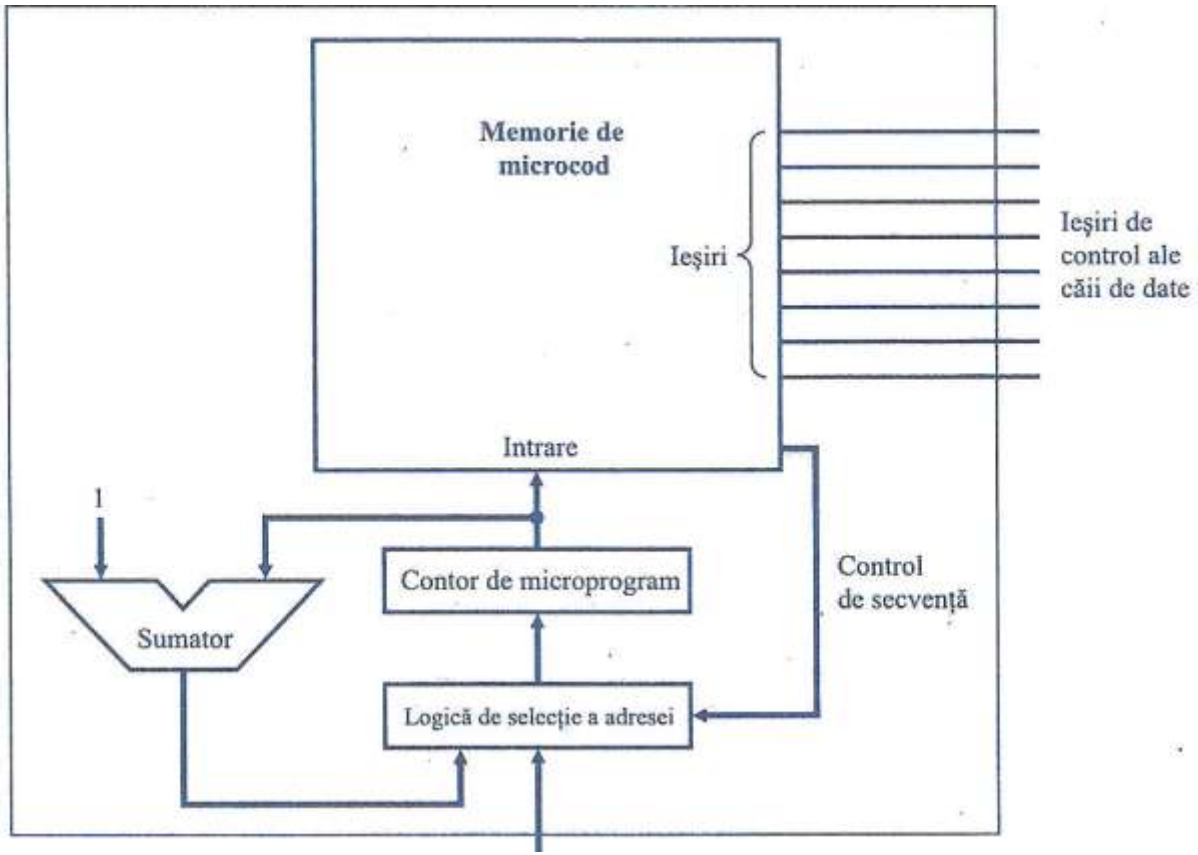
Etch	Ctrl UAL	SRC1	SRC2	Ctrl Reg	Mem	Ctrl ScriePC	Secv
BEQ1	Subt	A	B			ALUOut - cond	Fetch

De remarcat că avem doar o singură microinstrucțiune. **DE CE ?**

Instrucțiunea de salt

Etch	Ctrl UAL	SRC1	SRC2	Ctrl Reg	Mem	Ctrl ScriePC	Secv
JUMP1						JUMP address	Fetch

Implementarea controlului microcodului



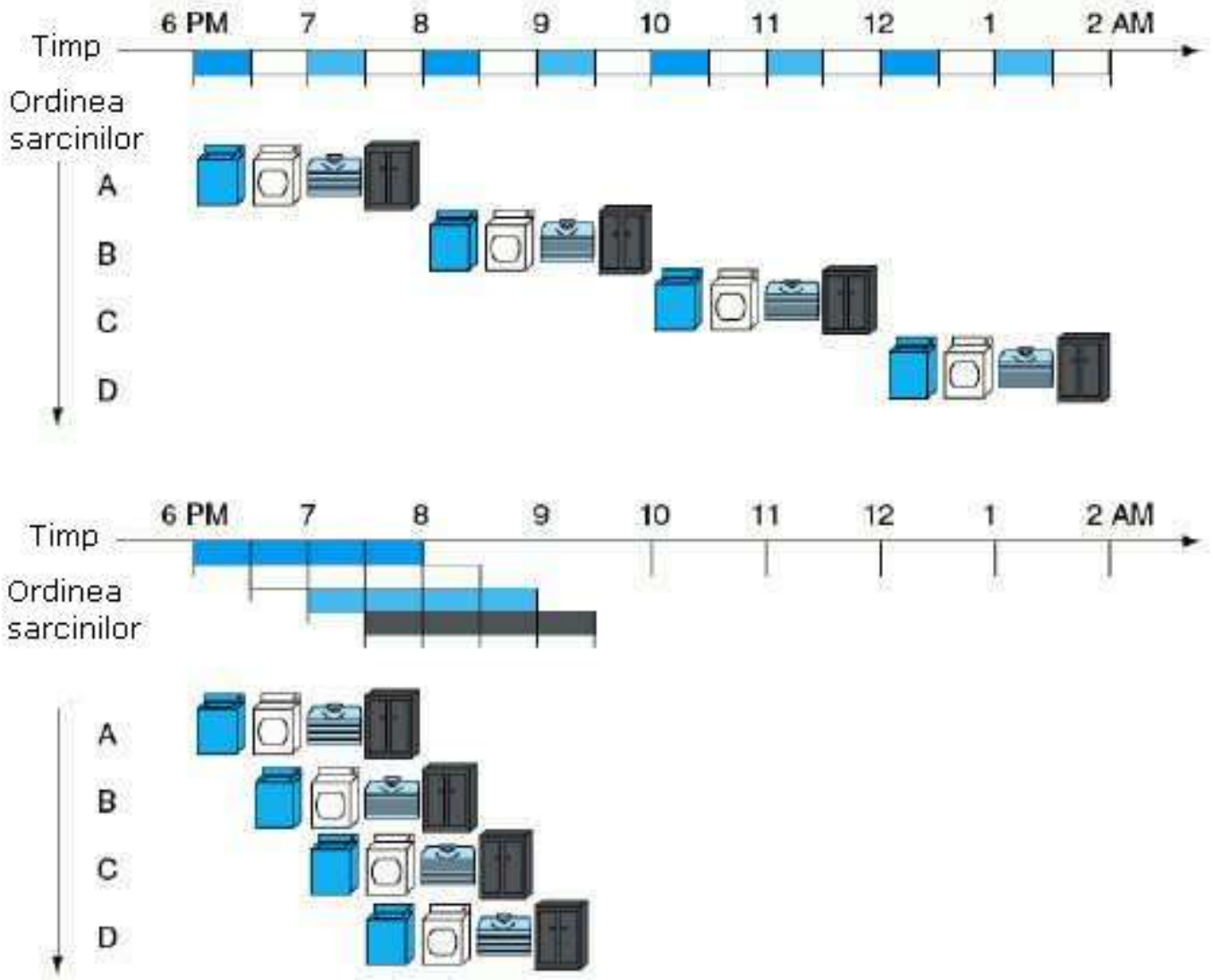
Unitate explicită pentru incrementare – calculează starea secvențială următoare implicită

Memoria de microcod poate fi doar citită

PIPELINE

Introducere

Pipeline – execuția mai multor instrucțiuni se suprapune în timp



Etajul 1 – extragerea instrucțiunii din memorie

Etajul 2 – citirea registrelor în timp ce instrucțiunea este decodificată

Etajul 3 – executarea operației sau calcularea adresei

Etajul 4 – accesul la un operand din memorie

Etajul 5 – scrierea rezultatului într-un registru

Timpul între instrucțiuni pipeline = Timpul între instrucțiuni fără pipeline / nr de etaje

Pipeline-ul îmbunătățeşte performanța prin creşterea productivității instrucțiunilor, nu prin micşorarea timpului de execuție al unei instrucțiuni individuale

OBSERVAȚII

Instrucțiunile au aceeaşi lungime => extragerea instrucțiunilor şi decodificarea lor se poate face în doar două etaje de pipe.

Câpurile registrelor sursă sunt localizate în acelaşi loc în cadrul instrucțiunii => în etajul 2 putem începe citirea fişierului de registre în acelaşi timp în care hardware-ul determină instrucțiunea ce a fost extrasă.

Operanzii din memorie apar numai în instrucțiunile de încărcare şi memorare => putem utiliza etapa de execuție pentru calcularea adresei de memorie, iar accesul la memorie se efectuează în etapa următoare.

Operanzii trebuie să fie aliniați în memorie => data poate fi transferată între procesor şi memorie într-o singură etapă pipeline.

HAZARDURI PIPELINE

Evenimentul în urma căruia instrucțiunea următoare nu poate fi executată în următorul ciclu de ceas se numeşte ***hazard***.

HAZARD SRUCTURAL

Hardware-ul nu poate suporta combinația de instrucțiuni pe care dorim să le executăm în acelaşi ciclu de ceas.

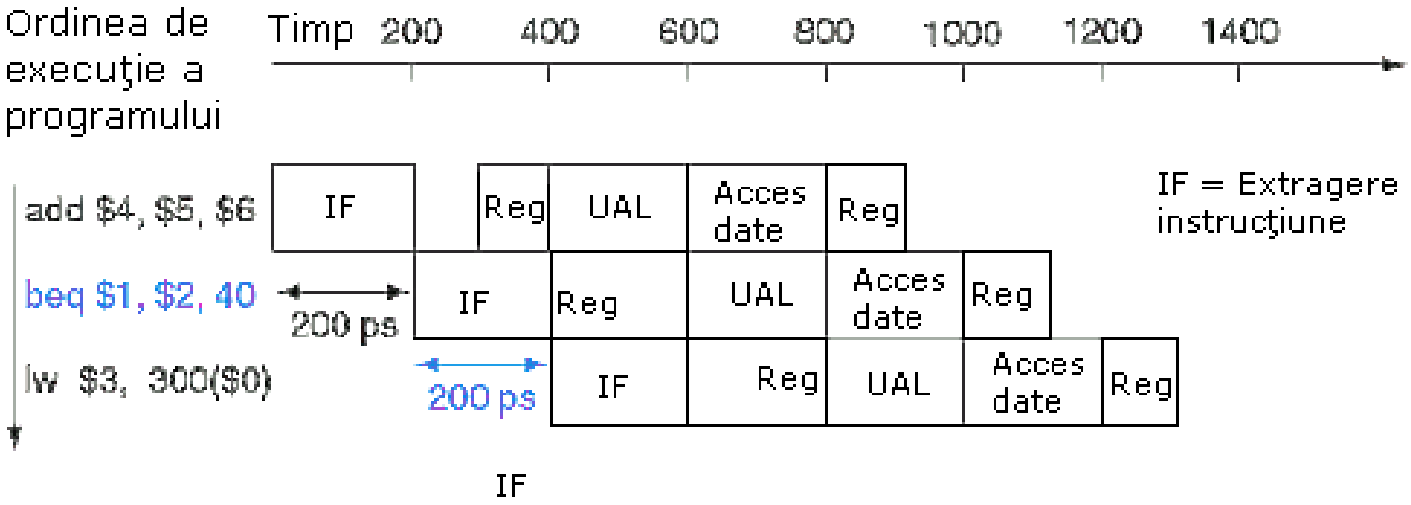
HAZARD DE CONTROL

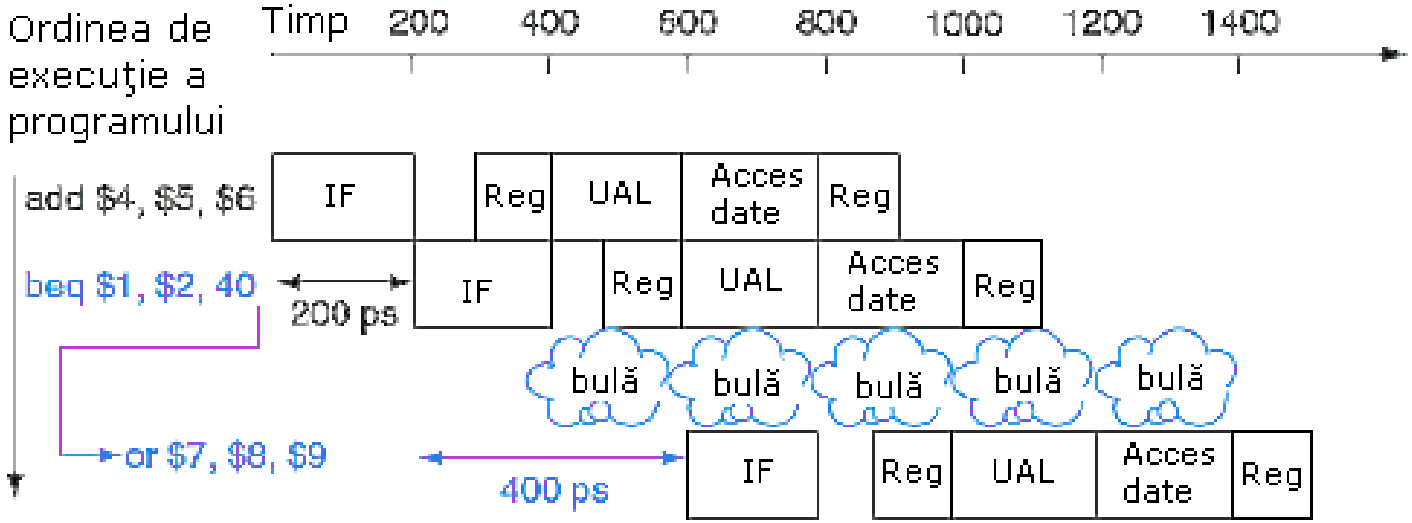
Apare din necesitatea de a lua o decizie pe baza rezultatelor unei instrucțiuni, în timp ce altele sunt în execuție

SOLUȚII

1. Staționarea – primul ciclu se va executa secvențial şi apoi se va trece la execuția pipeline – soluție viabilă în practică dar prea lentă în cazul pipe-urilor lungi

2. Predicția – se presupune că ramificațiile nu vor reuşi => că doar în cazul reuşitelor pipe-ul va staționa.





Cursul următor – pipeline superscalar şi pipeline dinamic

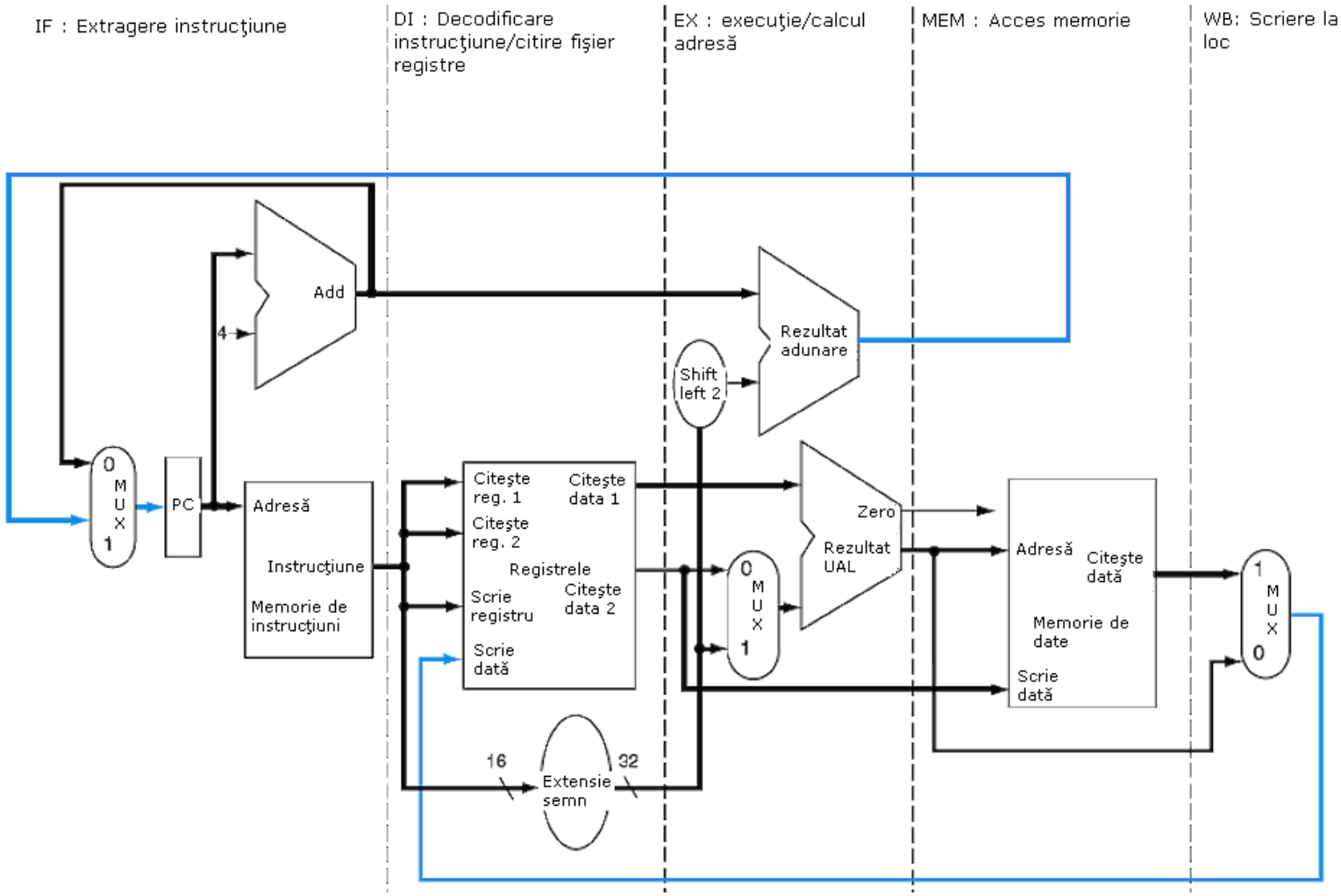
CONCLUZII

1. Pipeline-ul creşte numărul instrucțiunilor executate simultan şi viteza cu care instrucțiunile sunt începute şi terminate

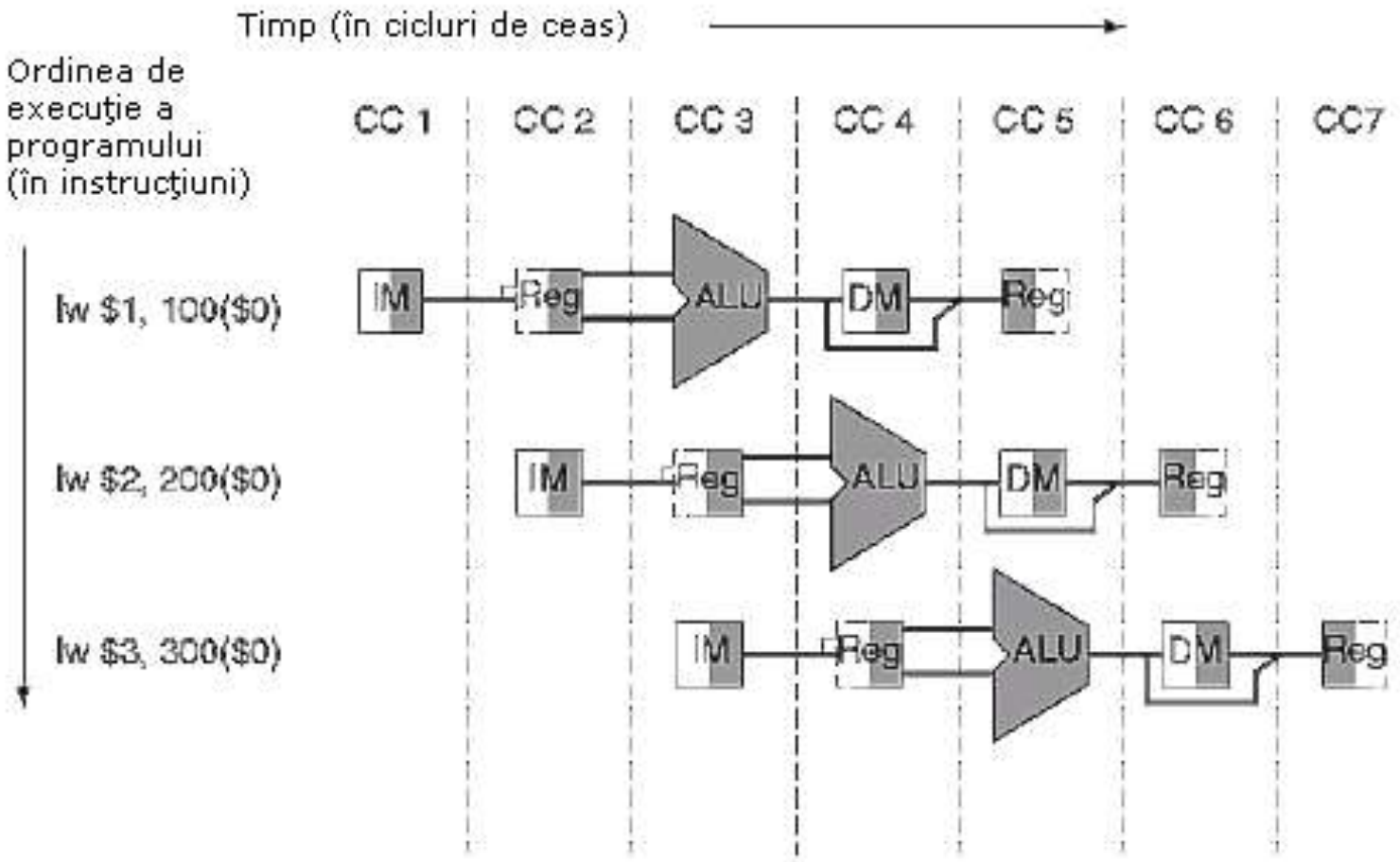
2. Execuția pipeline nu reduce timpul necesar derulării unei instrucțiuni individuale

Calea de date pipeline
- continuare -

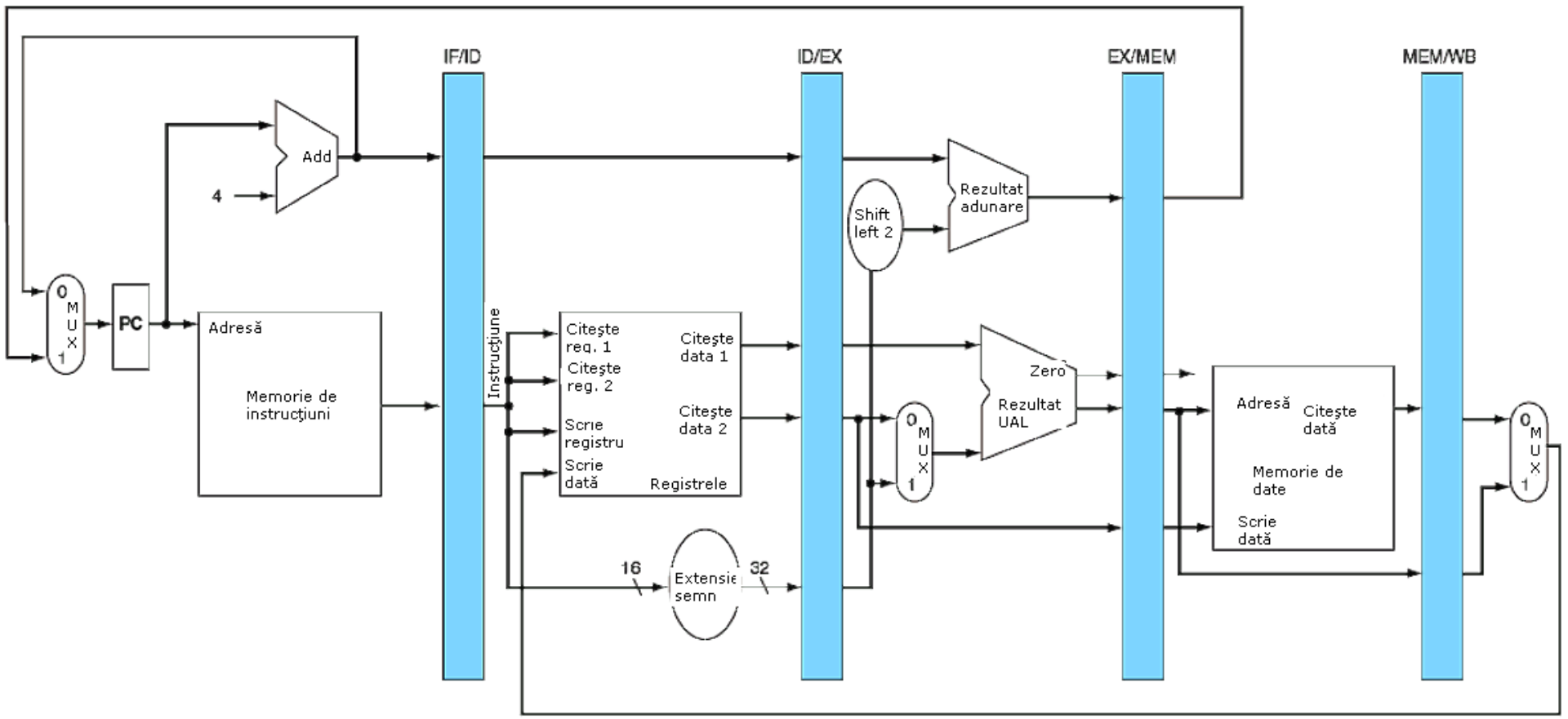
Calea de date cu un singur ciclu de ceas



Excepţii: actualizarea PC-ului şi pasul de rescriere



Fiecare instrucţiune are propia sa cale de date;
MI = memoria de instrucţiuni + PC-ul din etapa extragerii instrucţiunii
REG = fişierul de registre + unitatea de extindere a semnului din etapa DI



Registre pipeline separă fiecare etapă de pipe

EI/DI – separă etapele de extragere a instrucţiunii şi de decodificare a instrucţiunii

NU EXISTĂ REGISTRE LA SFÂRŞITUL ETAPEI DE RESCRIERE

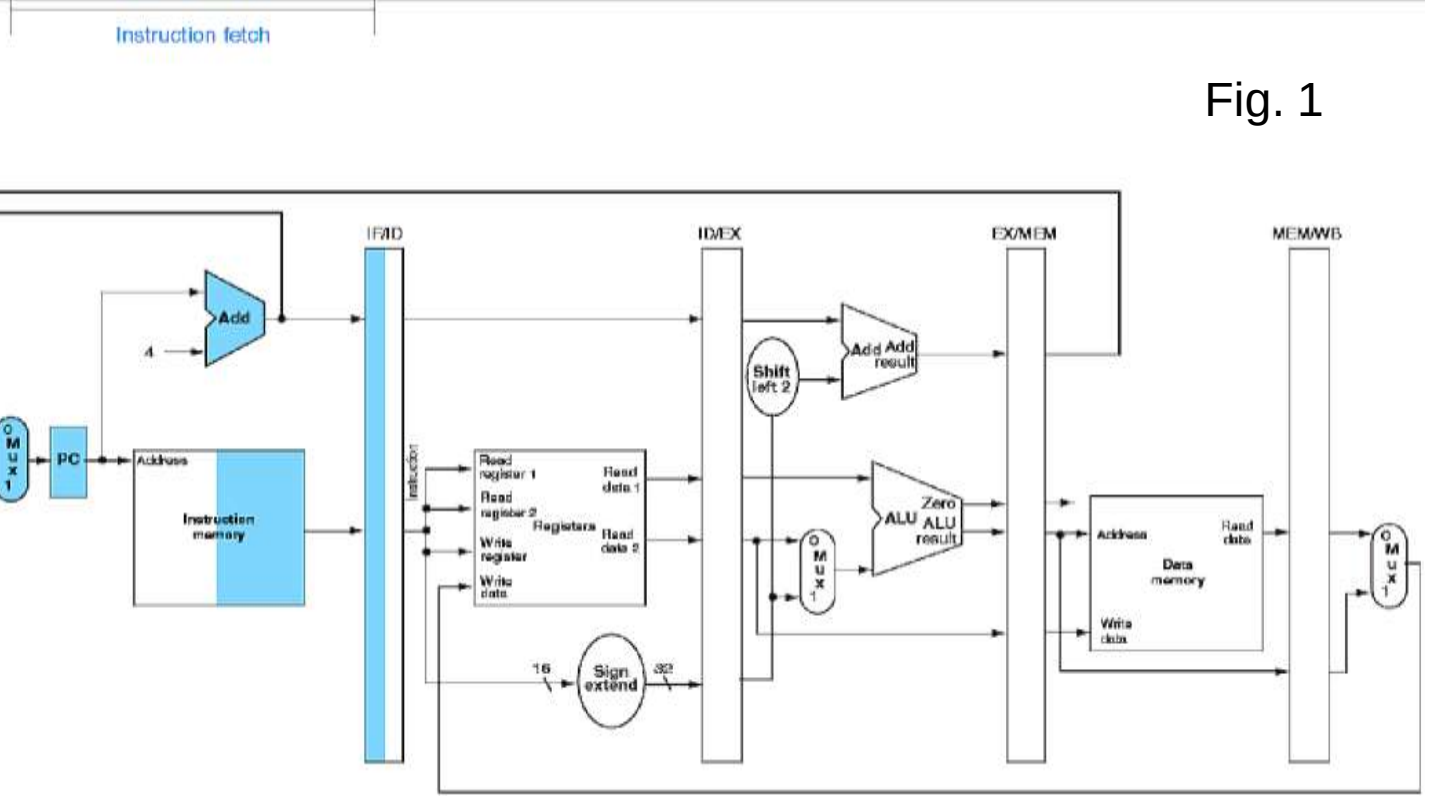


Fig. 1

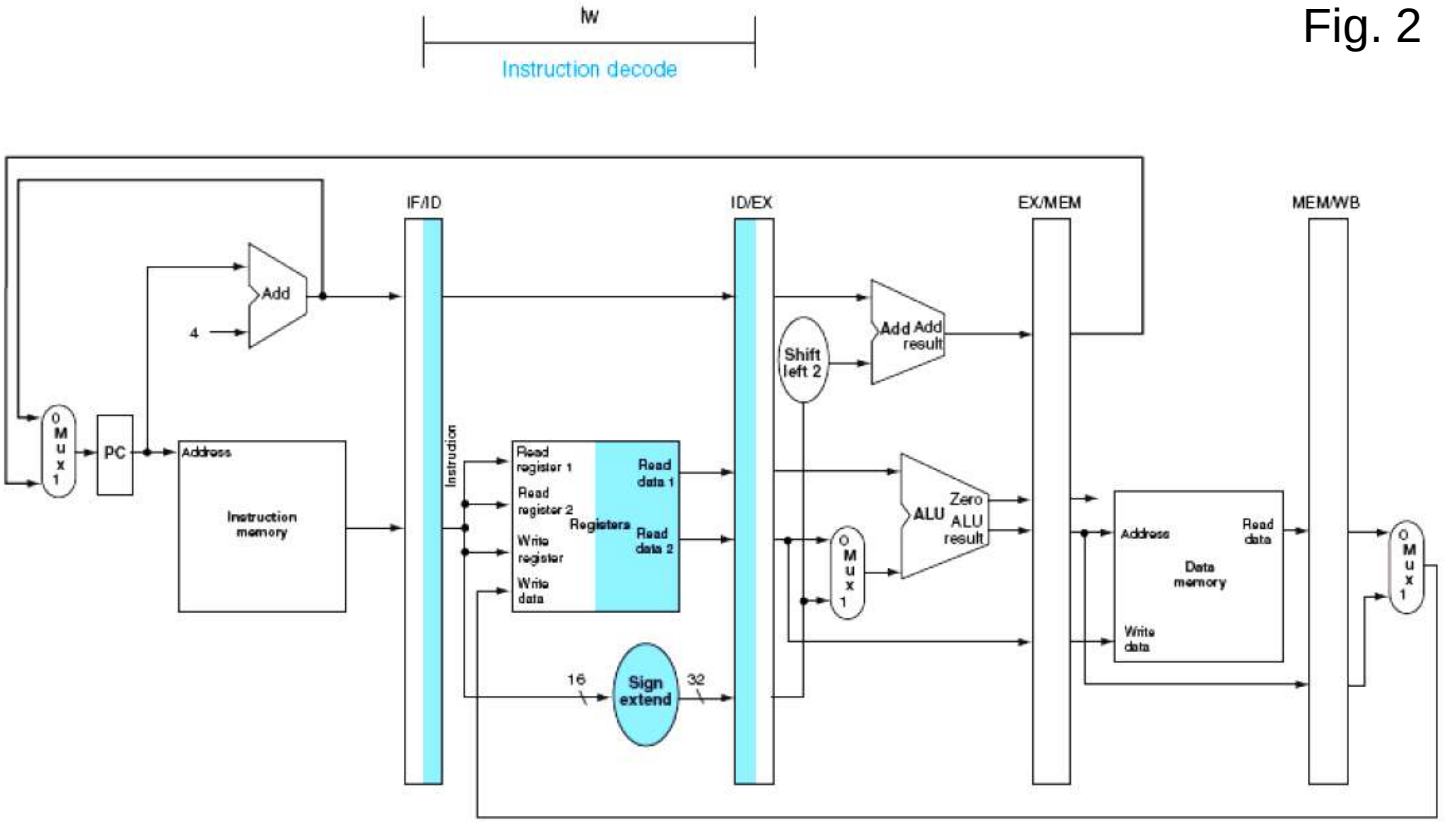
ÎNCĂRCAREA

Extragerea instrucţiunii – instrucţiunea este citită din memorie (folosind PC) şi este pusă în registrul EI/DI

Adresa lui PC este incrementată cu 4 şi rescrisă în PC. Această adresă este salvată în EI/DI în vederea folosirii ei ulterioare – spre exp. urmează instrucţiunea **beq**.

Calculatorul NU ştie ce instrucţiune este extrasă.

Fig. 2

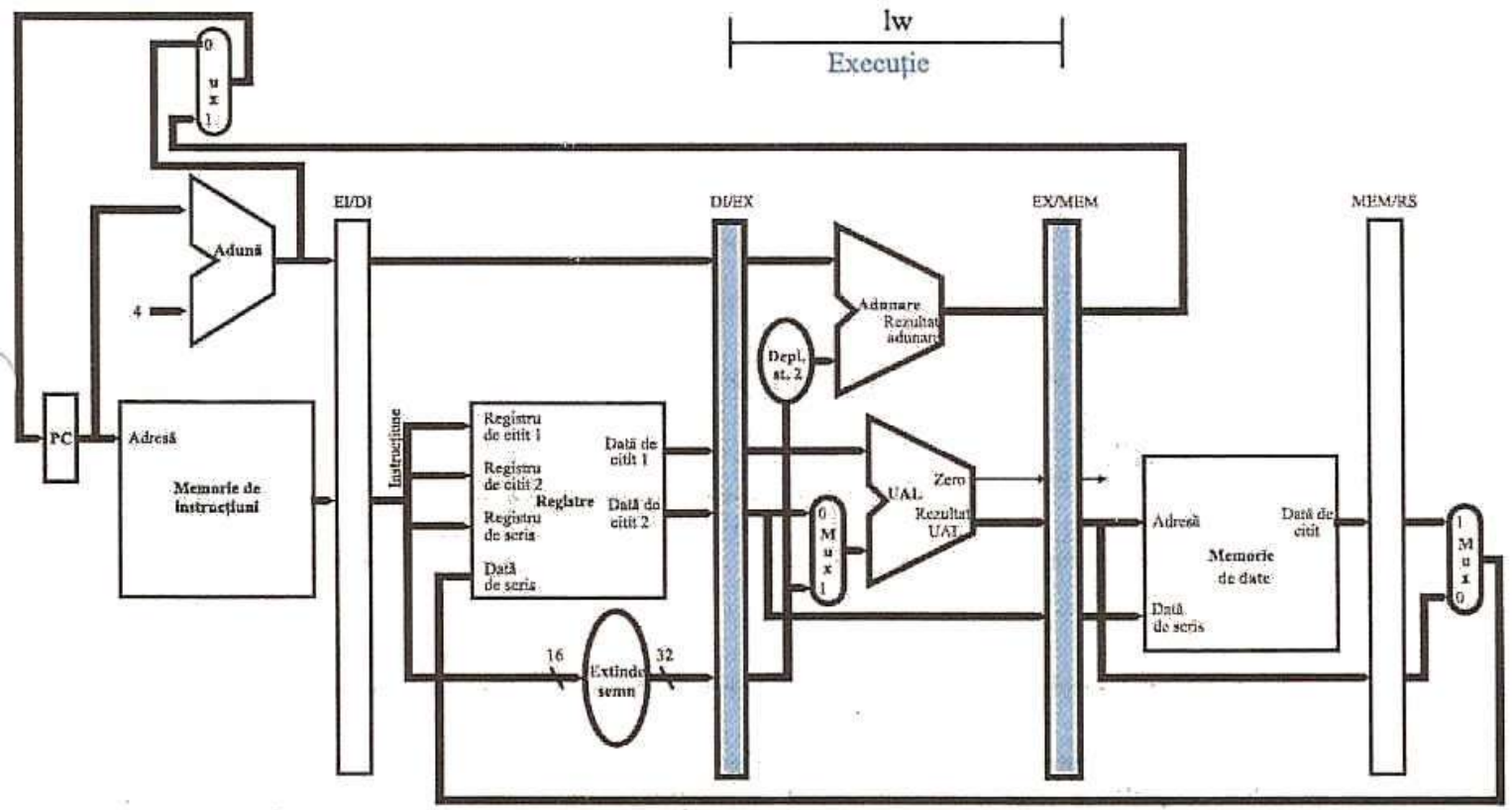


ÎNCĂRCAREA

Decodificarea instrucţiunii – registrul EI/DI furnizează câmpul imediat de 16 biţi căruia i se extinde semnul până la 32 de biţi, şi numerele registrelor pentru citirea celor două registre.

Aceste valori împreună cu adresa PC incrementată se regăsesc în DI/EX

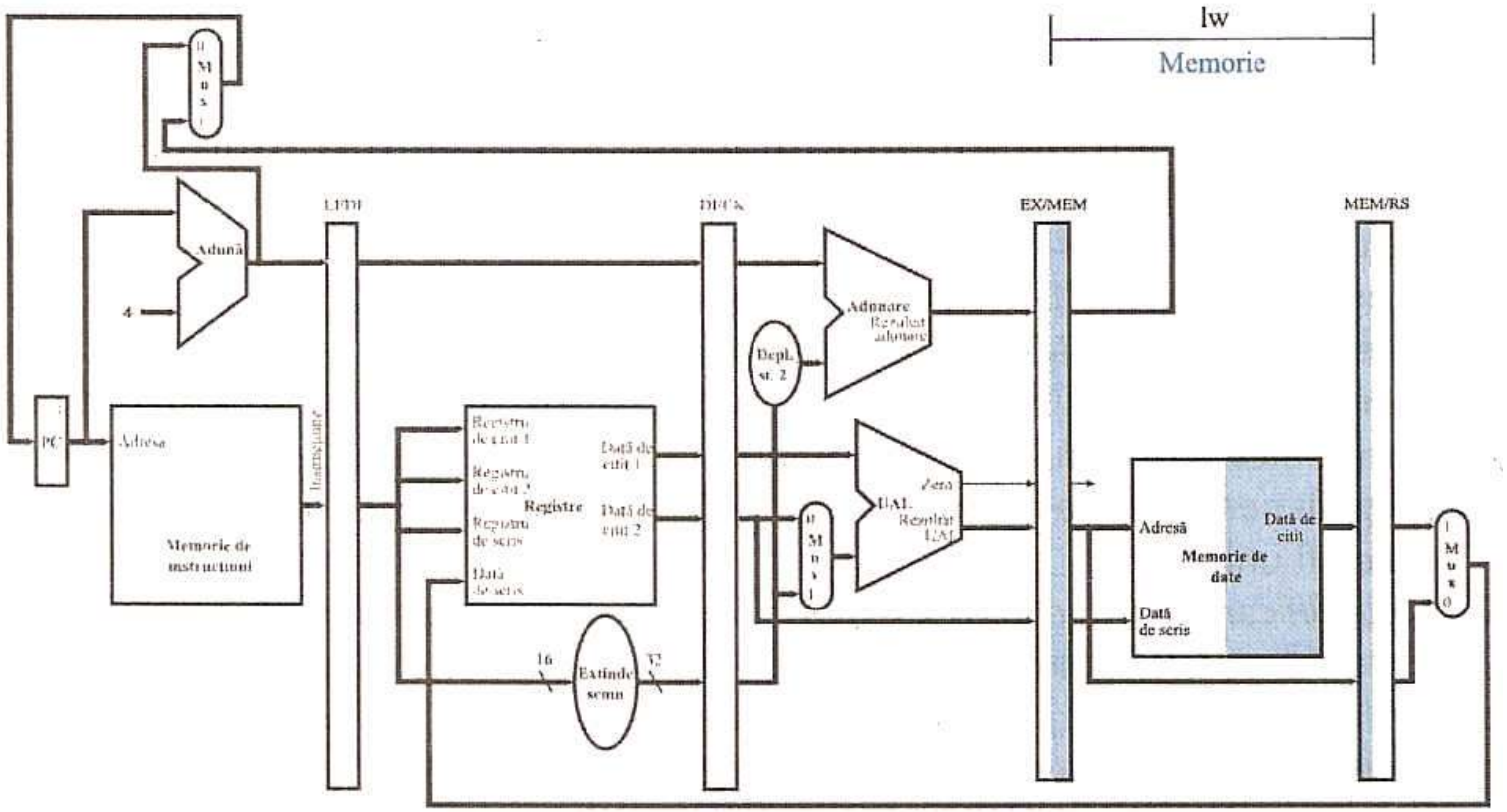
Sunt necesari toţi cei 3 operanzi ?



ÎNCĂRCAREA

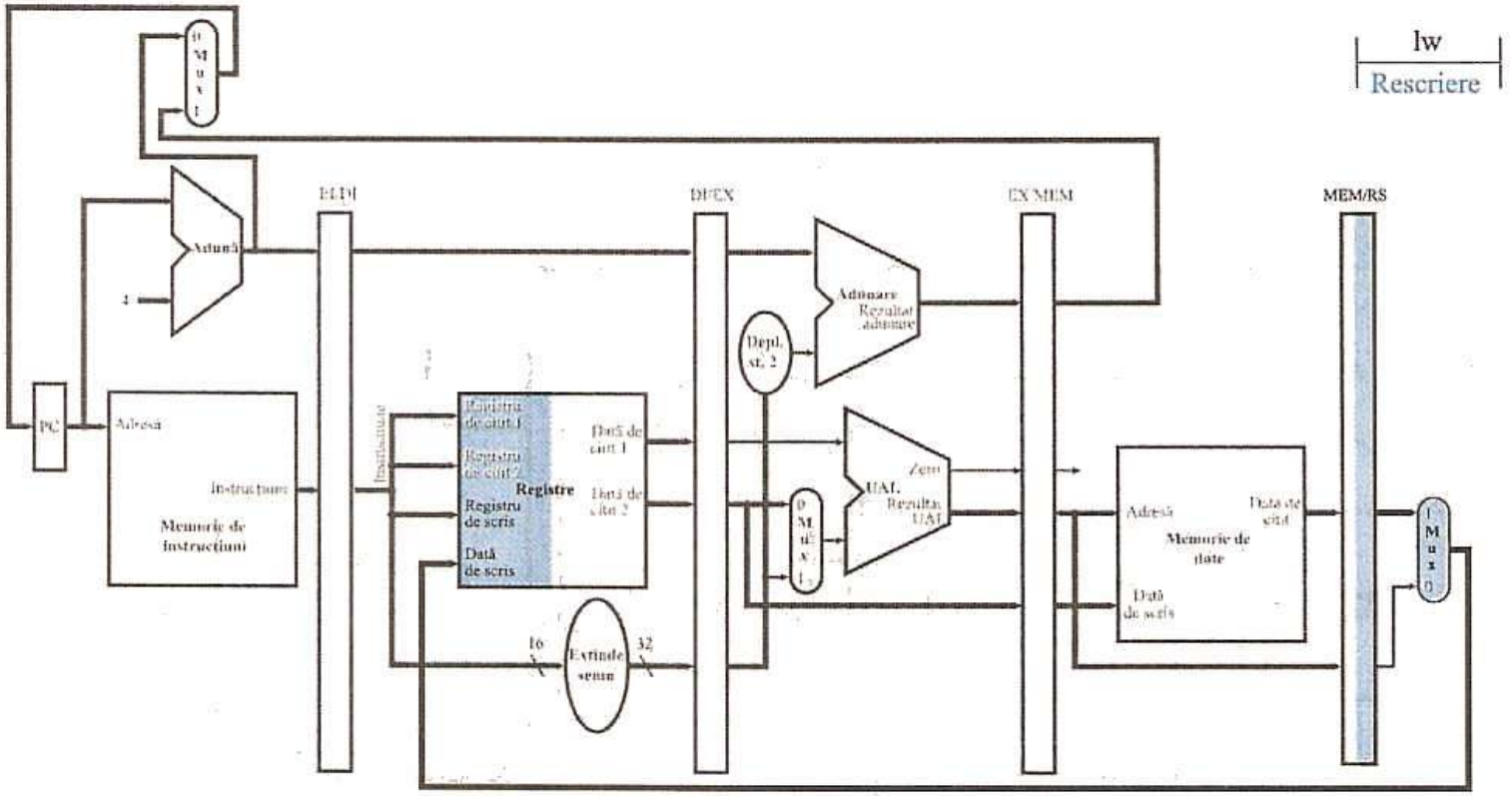
Execuţia instrucţiunii – instrucţiunea de încărcare citeşte din registrul pipeline EI/DI conţinutul registrului $t1 şi câmpul imediat cu semnul extins şi le adună cu ajutorul UAL-ului.

Suma se va pune în registrul pipeline EX/MEM.



ÎNCĂRCAREA

Accesul la memorie – instrucţiunea de încărcare citeşte memoria de datefolosind adresa din registrul pipeline EX/MEM şi încarcă data în registrul pipeline MEM/RS.



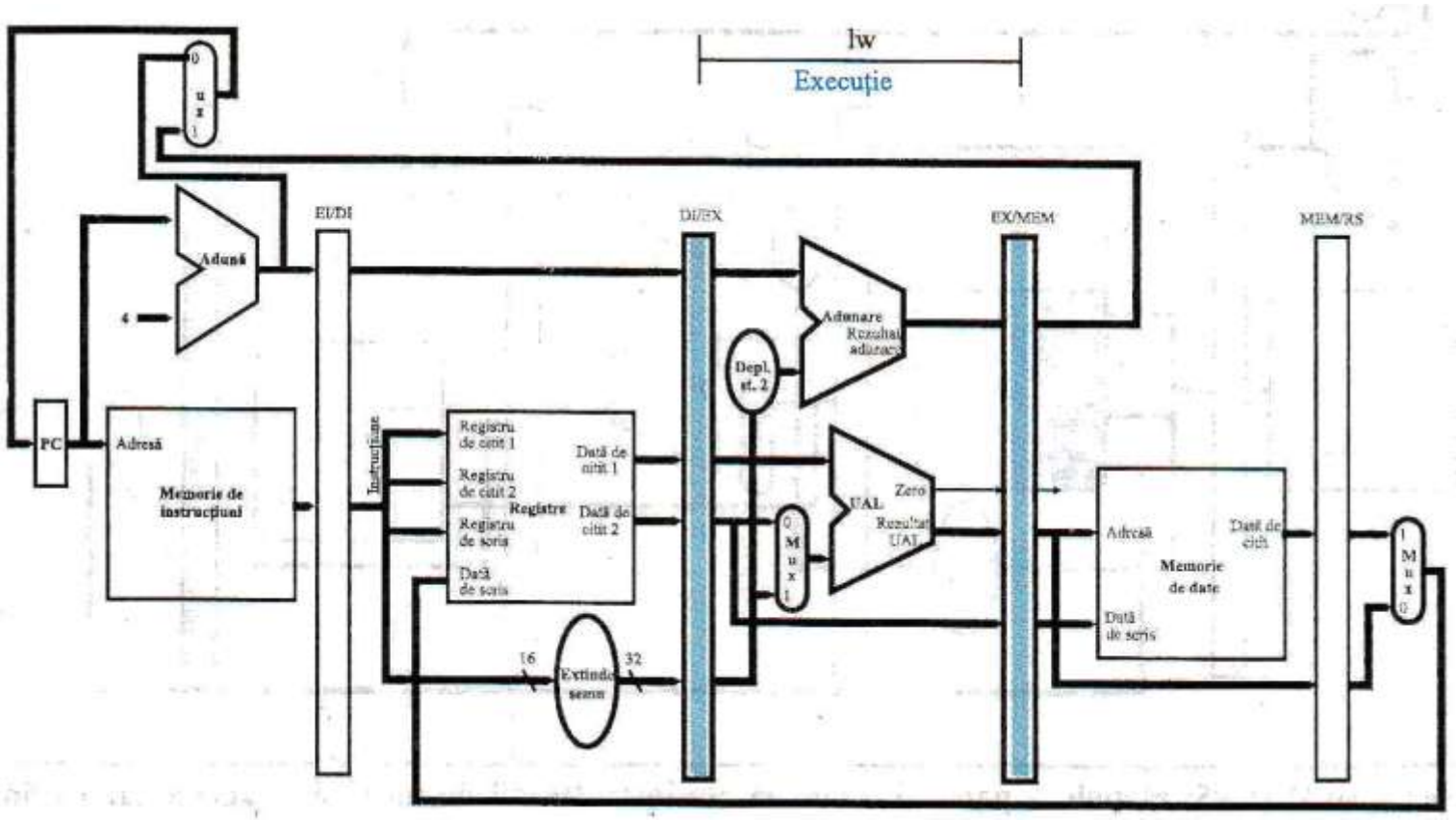
ÎNCĂRCAREA

Rescrierea fişierului de registre – citirea datei din registrul pipeline MEM/RS pe care o scrie în fişierul de registre.

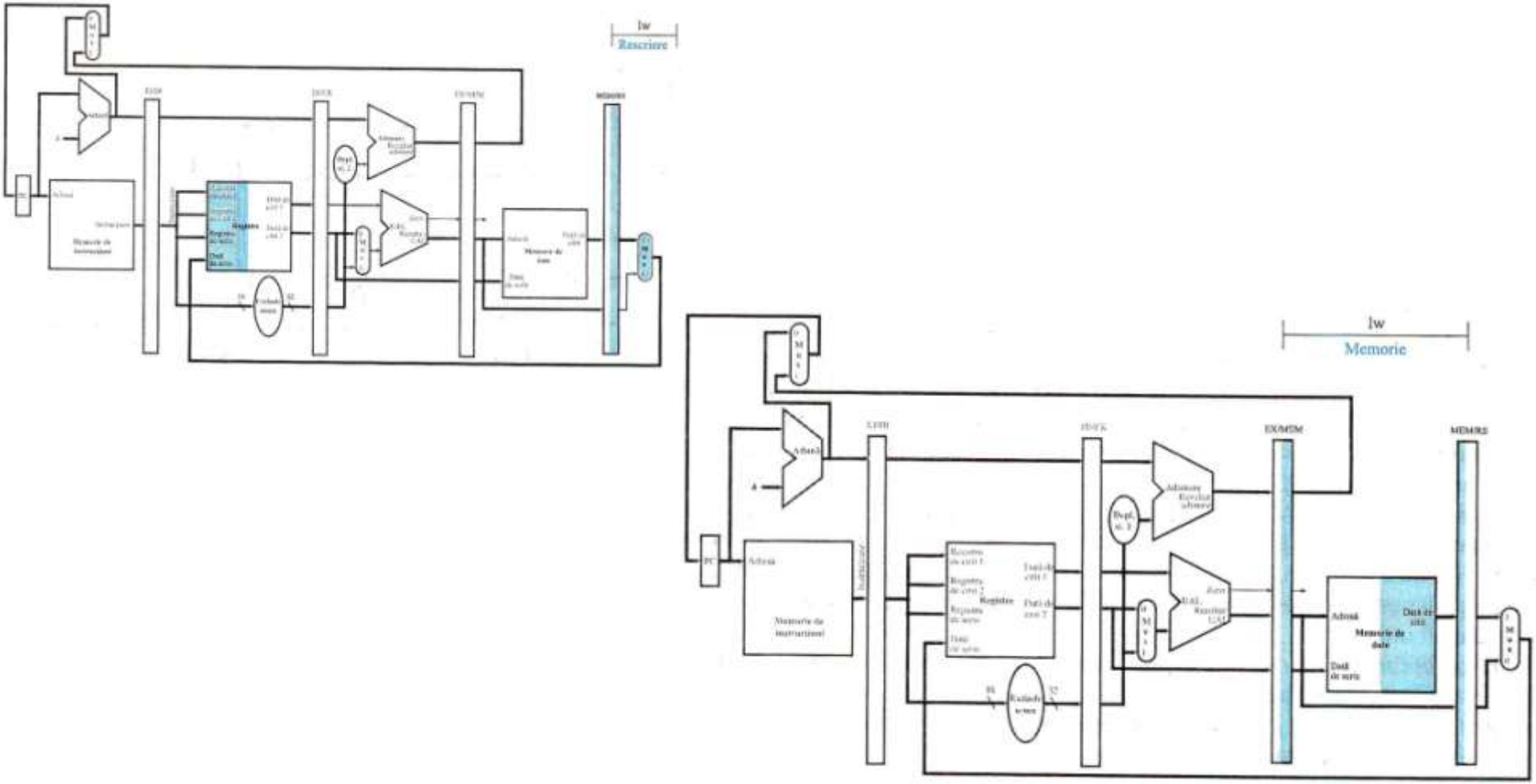
CONCLUZIE – orice informaţie necesară într-o etapă ulterioară de pipeline trebuie transmisă către etapa respectivă prin registrele pipeline.

INSTRUCŢIUNEA DE MEMORARE

1. EXTRAGEREA INTSRUCŢIUNII – instrucţiunea este citită din memorie folosindu-se adresa din PC, apoi este pusă în registrul pipeline EI/DI
2. DECODIFICAREA INTSRUCŢIUNII – respectă Fig. 2
3. EXECUŢIA INTSRUCŢIUNII – adună ce se citeşte din registrul EI/DI, conţinutul lui $t1, şi câmpul imediat cu semnul extins



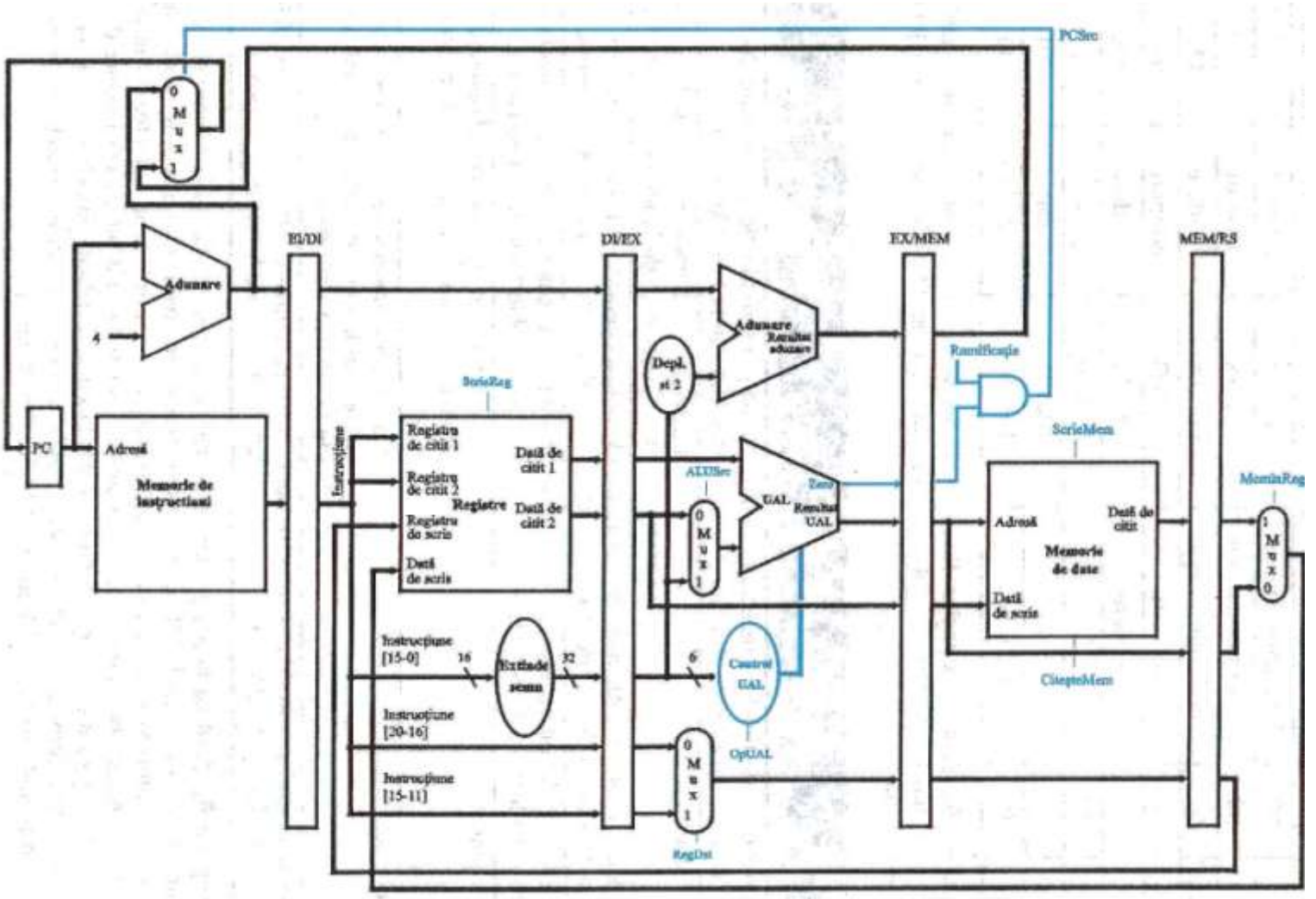
4. ACCESUL LA MEMORIE – citeşte memoria de date folosind adresa din registrul Pipeline EX/MEM şi încarcă data în registrul pipeline MEM/RS



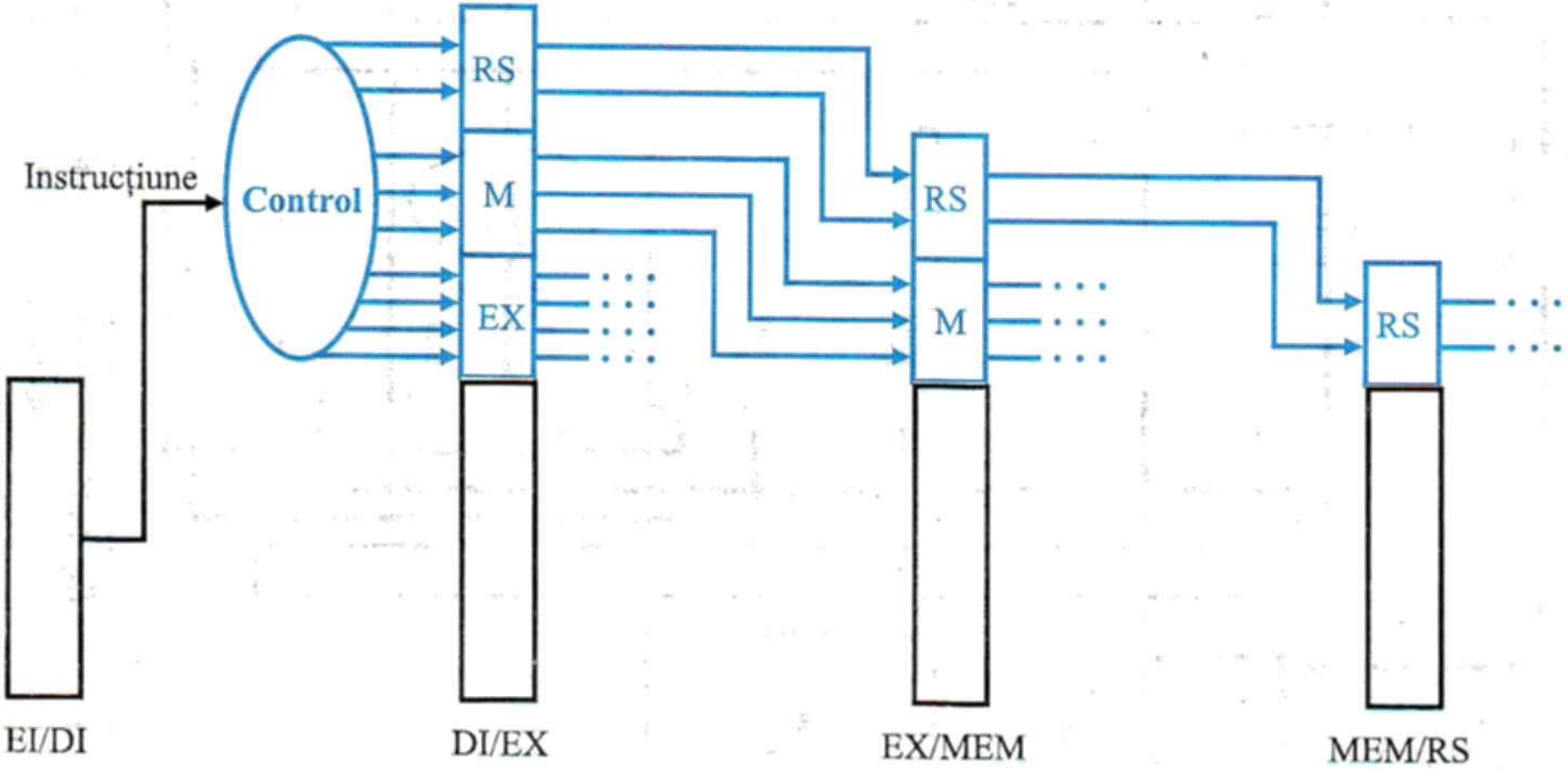
5. RESCRIEREA FIŞIERULUI DE REGISTRE – citirea datei din registrul pipeline MEM/RS pe care o scrie în fişierul de registre.

CONTROLUL PENTRU PIPELINE

Fiecare linie de control este asociată cu o componentă care este activă doar într-o Singură etapă pipeline.



Implementarea controlului



TEMĂ

Să se arate trecerea prin pipeline a următoarelor 5 instrucţiuni:

```
lw      $10, 20 ($1)
sub     $11, $2, $3
and     $12, $4, $5
or      $13, $6, $7
add     $14, $8, $9
```

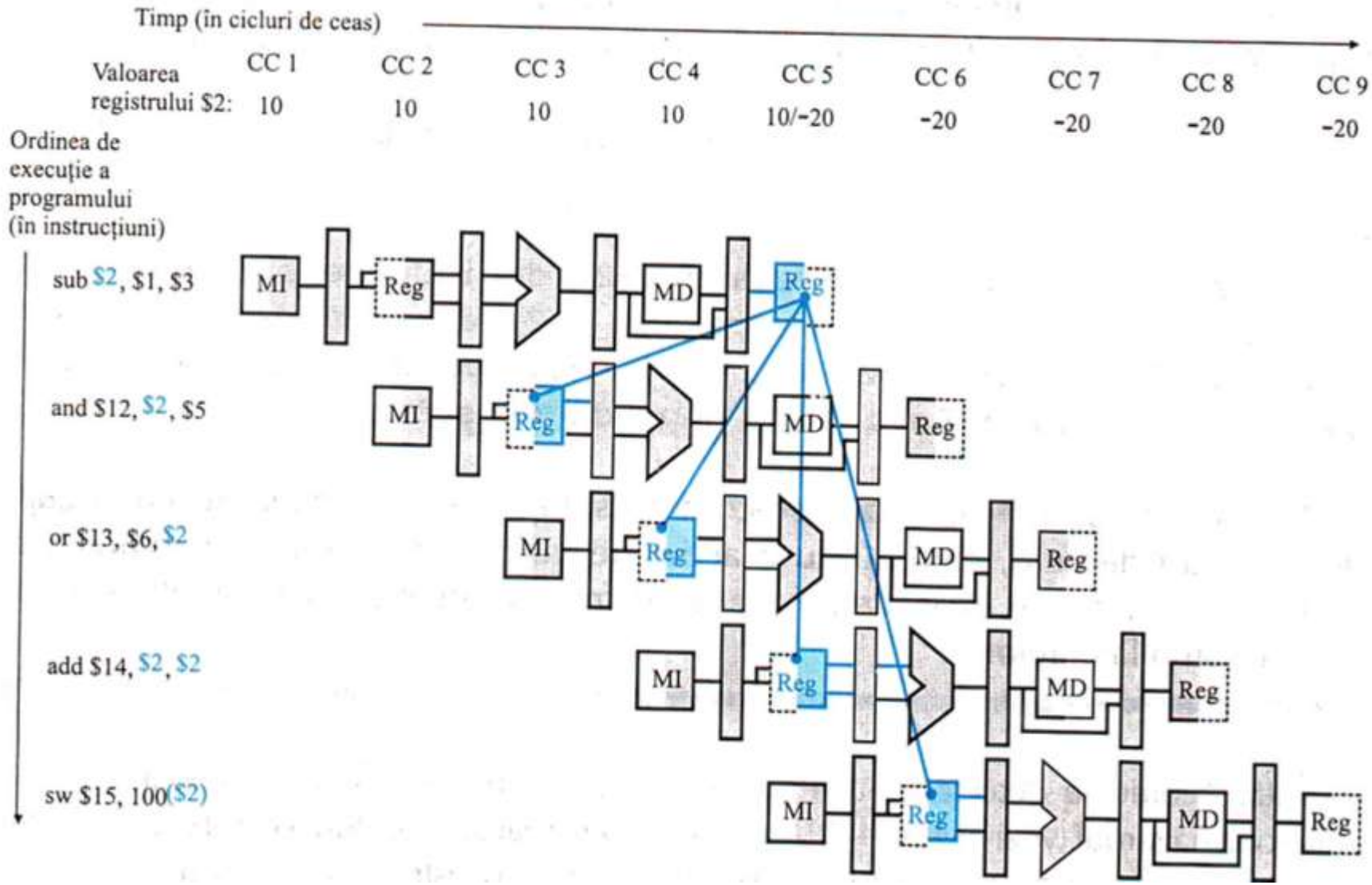
Să se eticheteze instrucţiunile din pipeline care precedă instrucţiunea lw sub forma

inainte <1>, inainte <2>, …..

şi instrucţiunile care urmează instrucţiunii add sub forma

dupa <1>, dupa <2>

HAZARDURILE DE DATE ŞI AVANSAREA



Soluția software – introducere nop-uri => cicluri de ceas în care nu se face nimic

Detecția hazardului

1a. EX/MEM.RegistruRd = DI/EX.RegistruRs

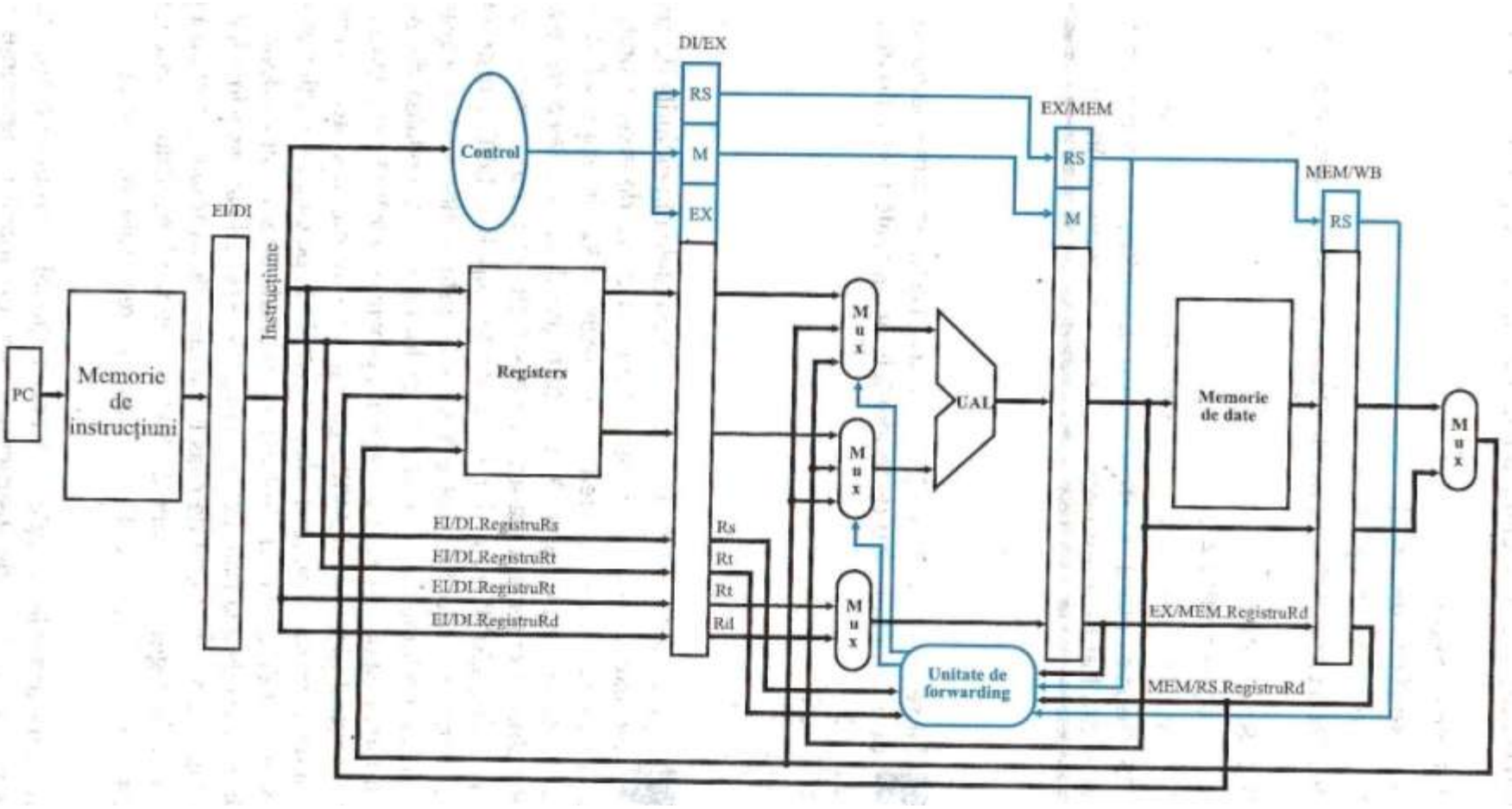
1b. EX/MEM.RegistruRd = DI/EX.RegistruRt

2a. MEM/RS.RegistruRd = DI/EX.RegistruRs

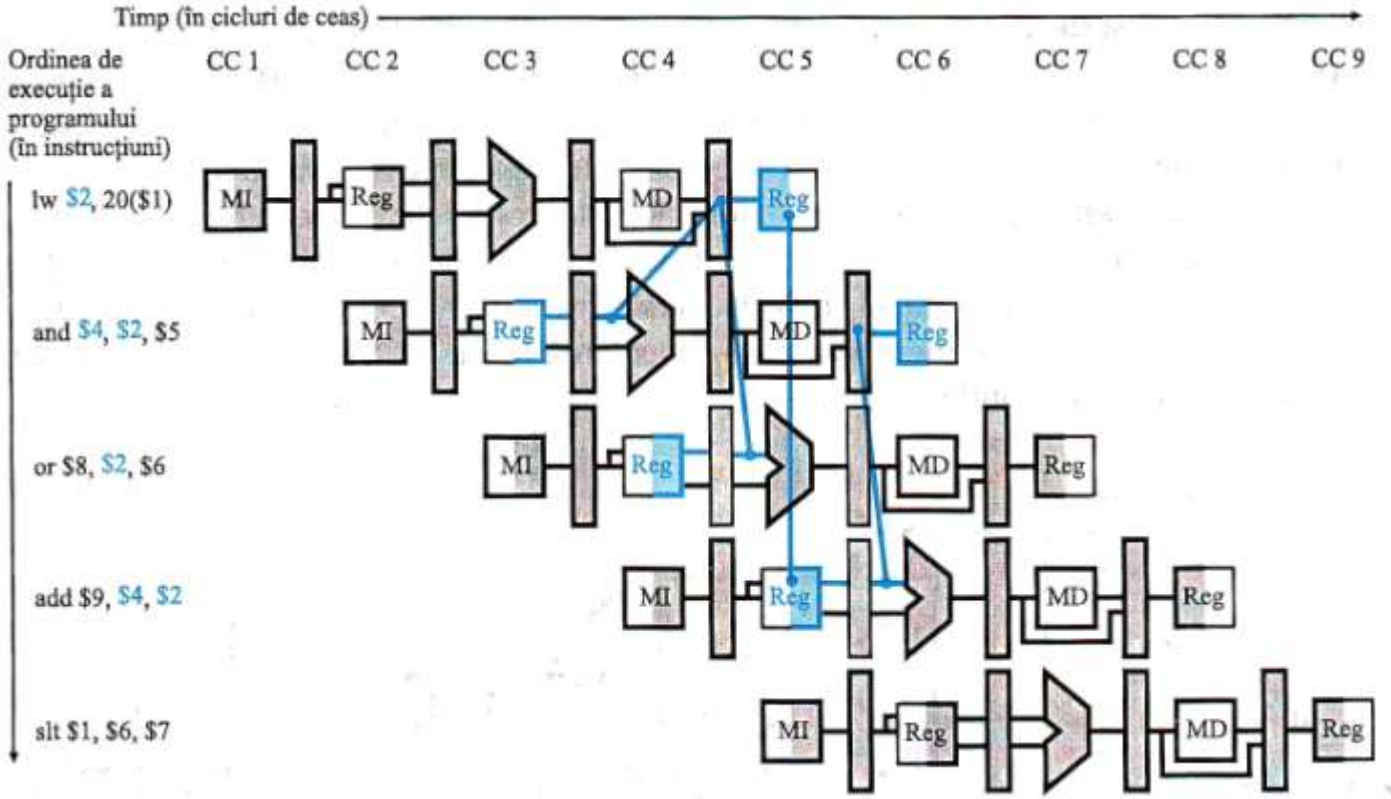
2b. MEM/RS.RegistruRd = DI/EX.RegistruRt

Dacă intrările UAL pot fi luate de la orice registru pipeline, nu numai de la EI/DI, atunci avansarea ar fi corectă.

Soluţia hardware



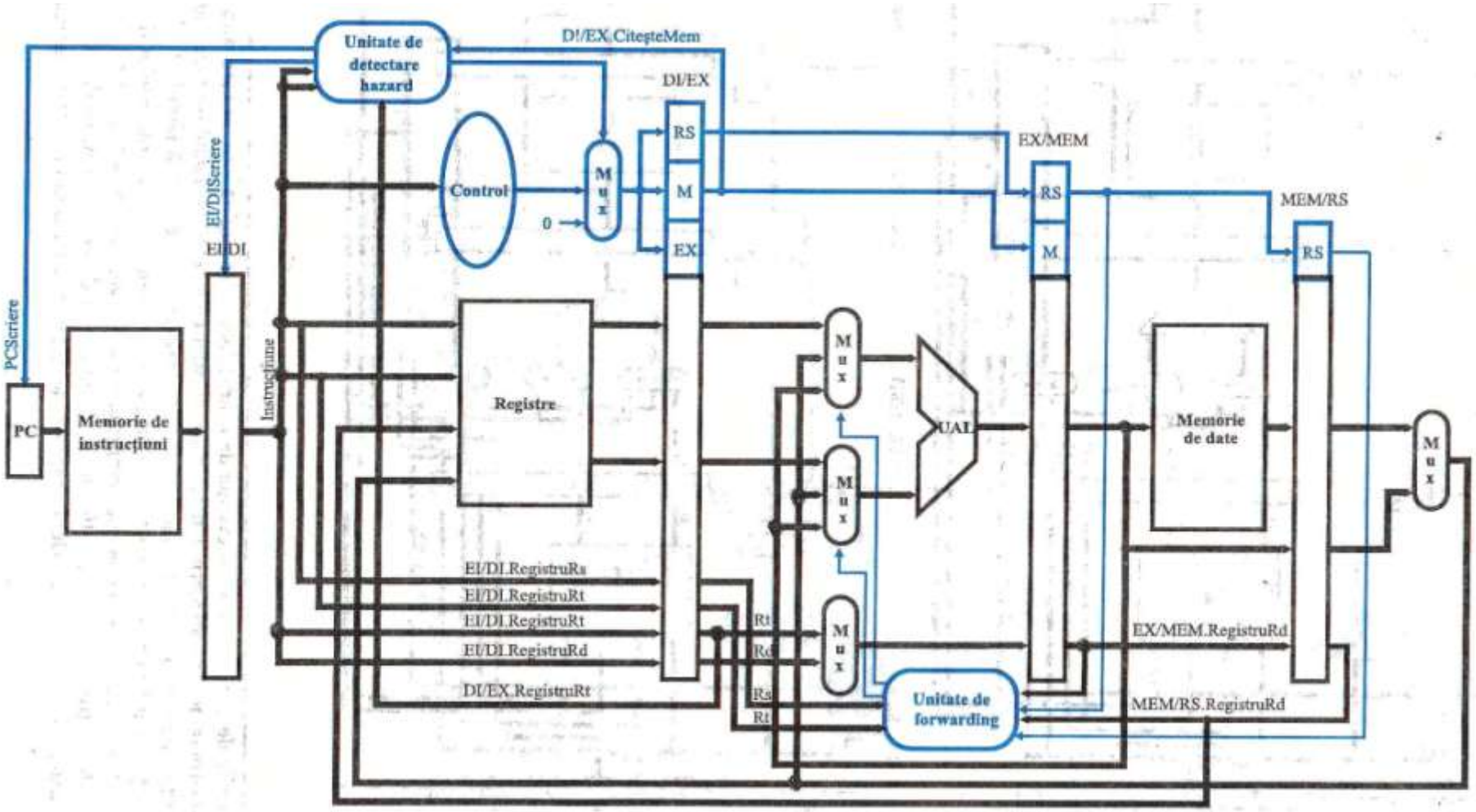
HAZARDURILE DE DATE ŞI STAŢIONĂRILE



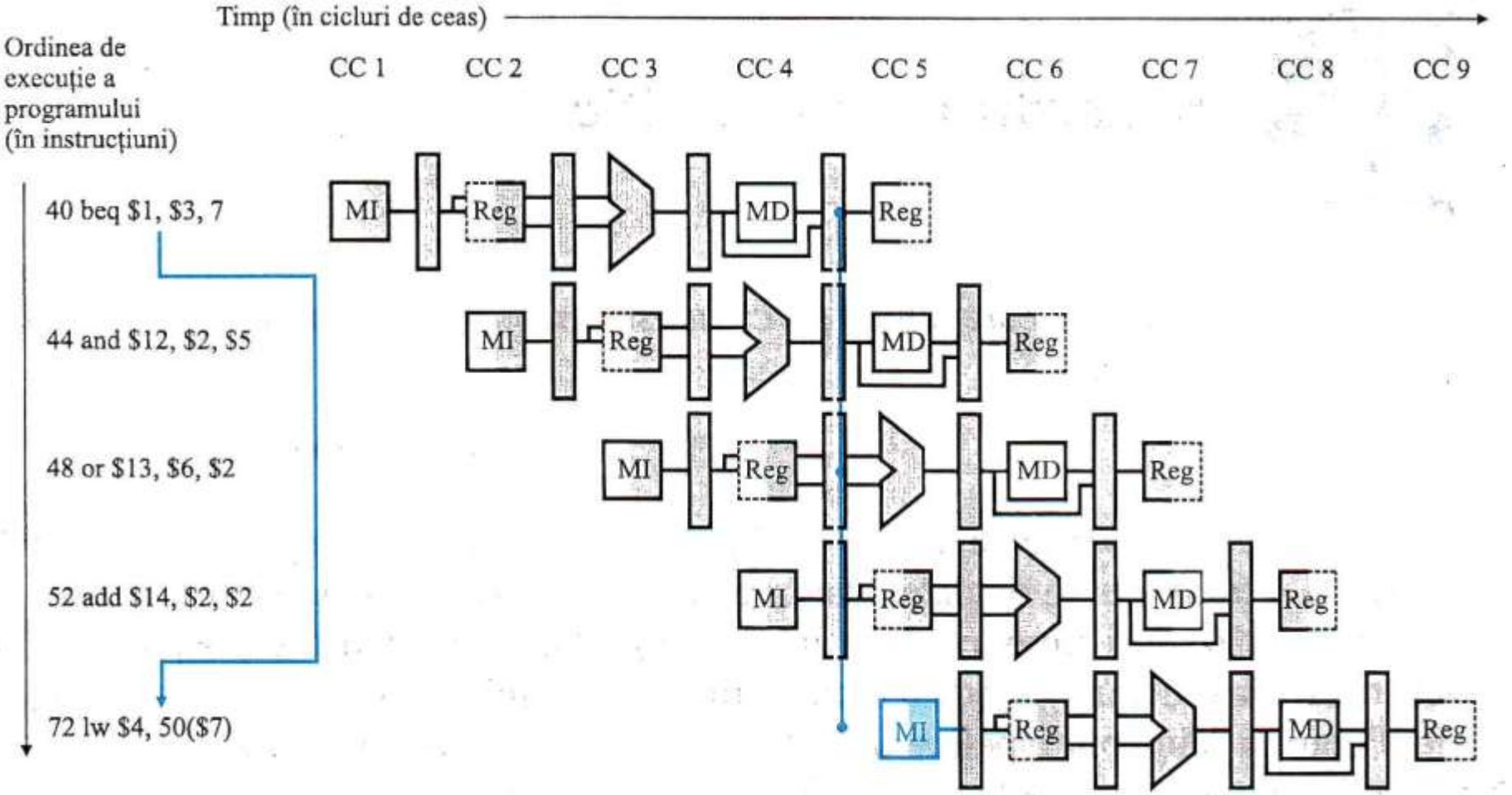
Pe lângă unitatea de avansare trebuie să existe o unitate de detectare a hazardului.

If (ID/EX.CiteşteMem and ((ID/EX.RegistruRt = IE/ID.RegistruRs) or
(ID/EX.RegistruRt = IE/ID.RegistruR)))
staţionare pipeline

Rezultatul final



Hazarduri de ramificaţie



Exploatarea ierarhiei de memorie

Un exemplu intuitiv

Un student aflat în biblioteca facultății are pe masă o serie de cărți

Subiectul căutat nu se regăseşte în cărțile aflate pe masă

Studentul se reîntoarce la rafturi şi extrage o nouă carte

Existența unui număr mare de cărți pe masă reduce timpul de căutare

Probabilitatea de căutare nu este aceeaşi pentru toate cărțile din bibliotecă

Un program nu accesează toată secțiunea sa de cod sau de date cu aceeaşi probabilitate.

Principiul localizării - stă la baza modului de operare a programelor

stabileşte faptul că programele accesează o porţiune relativ redusă a spaţiului lor de adrese la orice moment de timp

Localizarea temporală - localizare în timp - dacă se face referire la un anumit obiect, este posibil ca acesta să fie refrit din nou cât de curând

Localizarea spaţială - localizare în spaţiu - dacă se face referire la un anumit obiect din memorie, obiectele ale căror adrese sunt învecinate cu acesta tind să fie adresate cât de curând.

Principiul localizării temporare este folosit la implementarea memoriei unui calculator sub forma unei ierarhii de memorie.

O ierarhie de memorii poate fi alcătuită din mai multe niveluri, însă datele la un moment dat sunt copiate doar între două niveluri adiacente.

Unitatea minimă de informaţie care poate fi prezentă sau absentă într-o ierarhie de memorie se numeşte bloc.

Unitatea minimă de informaţie care poate fi prezentă sau absentă într-o ierarhie de memorie se numeşte bloc.

Regăsirea informaţiei necesare procesorului într-un bloc de memorie superior se numeşte **HIT**

Neregăsirea datelor pe nivelul superior se numeşte **MISS**

Rata de succes - fracţiunea din accesele la memorie ce au găsit datele în nivelul superior de memorie.

Rata de eşec = 1 - rata de succes fracțiunea din accesele la memorie care nu au găsit datele în nivelul superior de memorie.

Timpul de succes - reprezintă timpul necesar accesului la un nivel superior al ierarhiei de memorie, ce include şi timpul necesar determinării tipului de acces.

Penalizarea de eşec - reprezintă timpul necesar înlocuirii blocului din nivelul superior cu blocul corespunzător din nivelul inferior, incluzând şi timpul trimiterii acestui bloc către procesor.

Construirea sistemelor de memorie afectează:

1. modul în care SO-ul administrează memoria şi perifericele
2. modul în care compilatoarele generează codul
3. modul în care aplicaţiile folosesc maşina de calcul

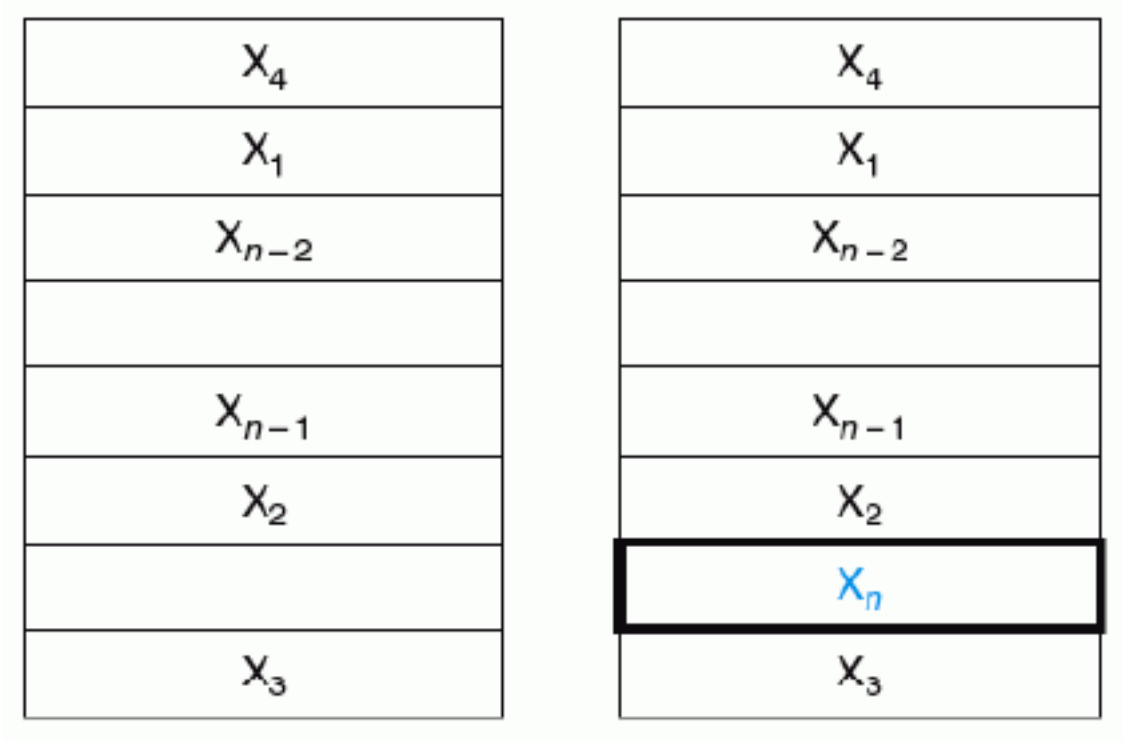
Principiu de bază

Programele prezintă localizare temporală cât şi localizare spațială.

Ierarhiile de memorie folosesc avantajul localizării temporale păstrând datele accesate recent cât mai aproape de procesor.
Ierarhiile de memorie folosesc avantajullocalizării spațiale prin mutarea blocurilor conţinând cuvinte învecinate din memorie în nivelurile superioare ale ierarhiei.

În anii ’60 s-a folosit cuvântul ***cache*** pentru a desemna nivelul ierarhiei de memorie aflat între UCP şi memoria principală.

PRINCIPIILE DE BAZĂ ALE MEMORIILOR CACHE



Inițial cuvântul de date X_n nu se găsește în cache

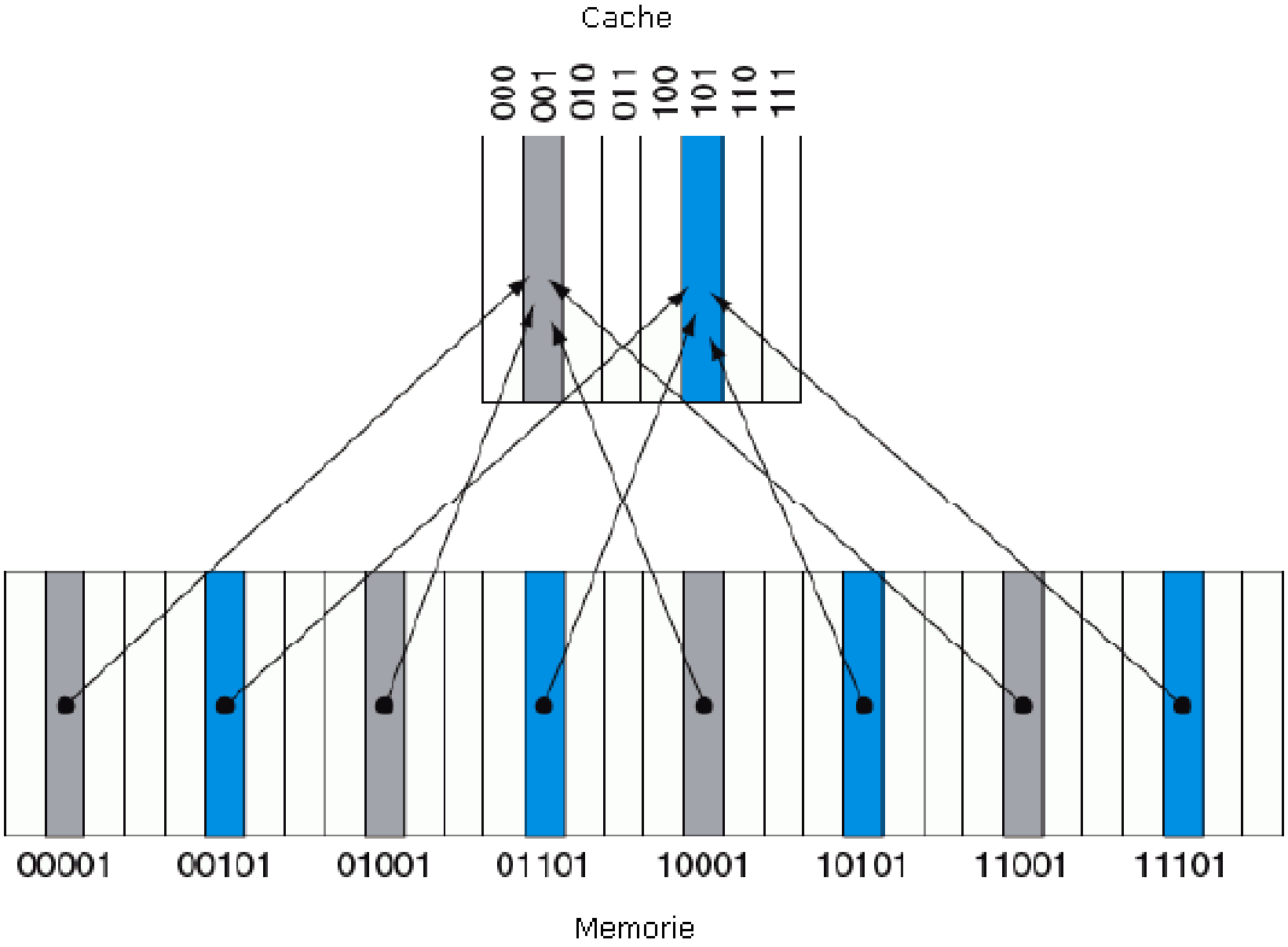
Cum se poate determina dacă o dată este în memoria cache ?

Dacă o dată există în memoria cache cum poate fi ea gasită ?

!!!!! Fiecare cuvânt poate fi memorat doar într-o anumită locație a memoriei cache => corespondență directă.

Corespondența dintre adrese și locațiile memoriei cache se determină astfel:

(Adresa blocului) modulo (numărul de blocuri din memoria cache)



Cum determinăm dacă o dată este în memoria cache ?

Cum determinăm dacă o dată este validă sau nu ?

Metoda cea mai des folosită este cea de a aduna un ***bit de validare*** pentru a indica dacă o locație conține o adresă validă.

Accesarea unei memorii cache cu corespondeţă directă

Cerere	Adresa	HIT / MISS	Blocul din cache
22	10110		
26	11010		
22	10110		
26	11010		
16	10000		
3	11		
16	10000		
18	10010		

Index	V	Marcaj	Date
0	N		
1	N		
10	N		
11	N		
100	N		
101	N		
110	N		
111	N		

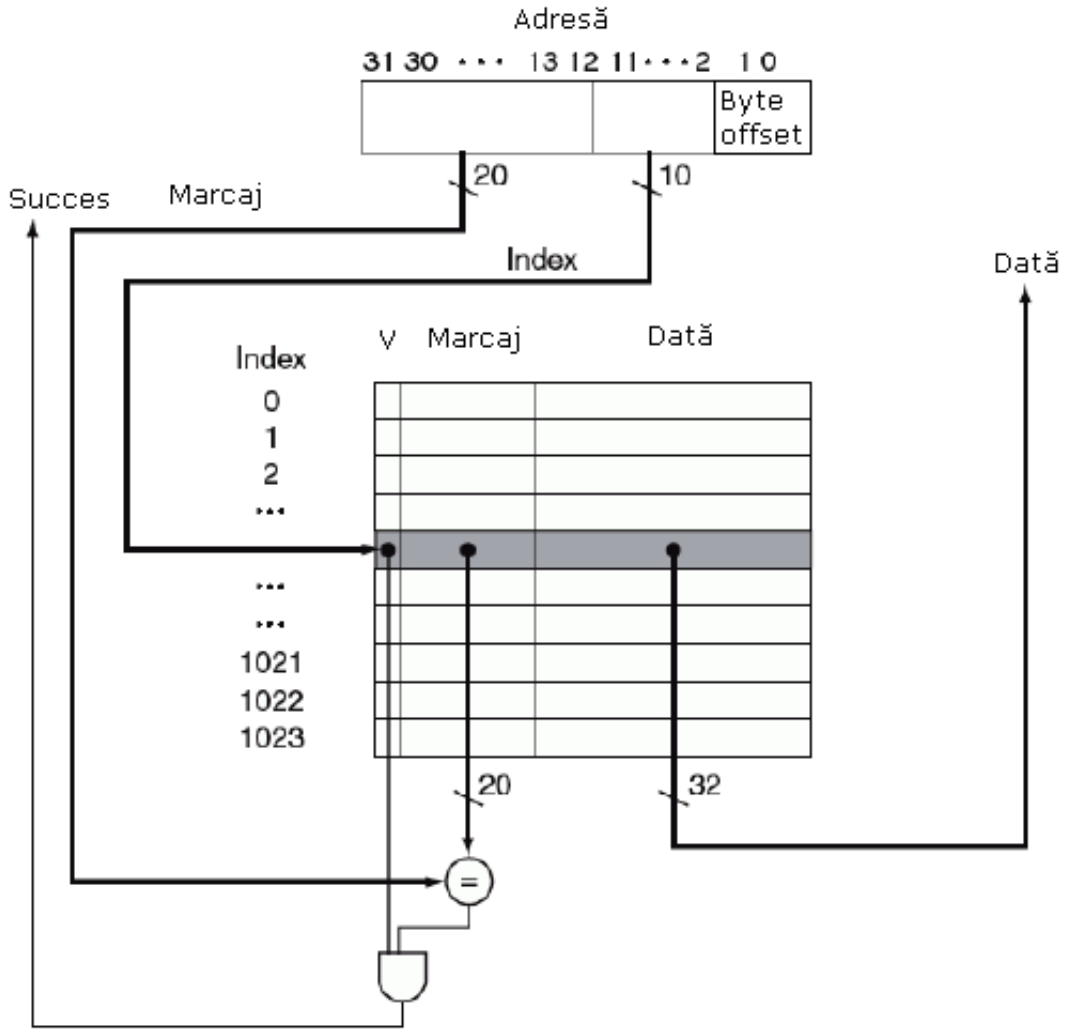
Index	V	Marcaj	Date
0	N		
1	N		
10	N		
11	N		
100	N		
101	N		
110	Y	10	mem (10110)
111	N		

Index	V	Marcaj	Date
0	N		
1	N		
10	Y	11	mem (11010)
11	N		
100	N		
101	N		
110	Y	10	mem (10110)
111	N		

Index	V	Marcaj	Date
0	Y	10	mem (10110)
1	N		
10	Y	11	mem (11010)
11	N		
100	N		
101	N		
110	Y	10	mem (10110)
111	N		

Acest tip de accesare permite folosirea principiului localizării temporale - cuvintele accesate recent le înlocuiesc pe cele accesate mai puțin recent.

Tag = Marcaj



1. indexul memoriei cache, folosit la selectarea blocului de memorie cache

2. câmpul marcajului, folosit la compararea cu valoarea din câmpul marcaj al memoriei cache

Numărul total de biţi dintr-o memorie cache cu corespondenţă directă este:

2^n * (dimensiunea blocului de memorie + dimensiune marcaj + dimensiunea câmpului de validare).

TEMĂ

Câţi biţi sunt necesari pentru o memorie cache cu corespondenţă directă având 64KB de date şi blocuri de 1 cuvânt, folosind adrese de 32 de biţi ?

Tratarea eşecurilor

a). Soluţia cea mai simplă presupune staţionarea UCP-ul, înghețând conţinutul tuturor registrelor.

O unitate de control separată tratează eşecul, aducând data din memoria principală în memoria cache

Execuţia este reluată începând cu ciclul care a cauzat eşecul.

Tratarea eşecului se face de către unitatea de control a procesorului şi de către o unitate de control separată ce iniţiează accesul la memorie şi aduce datele în memoria cache.

b). În cazul implementării pipeline tratarea eşecurilor la memoria cache este mai dificilă deoarece execuţia unor instrucţiuni trebuie continuată, în timp ce altele staţionează.

1). se trimite valoarea originală a PC-ului (PC-4) la memorie;

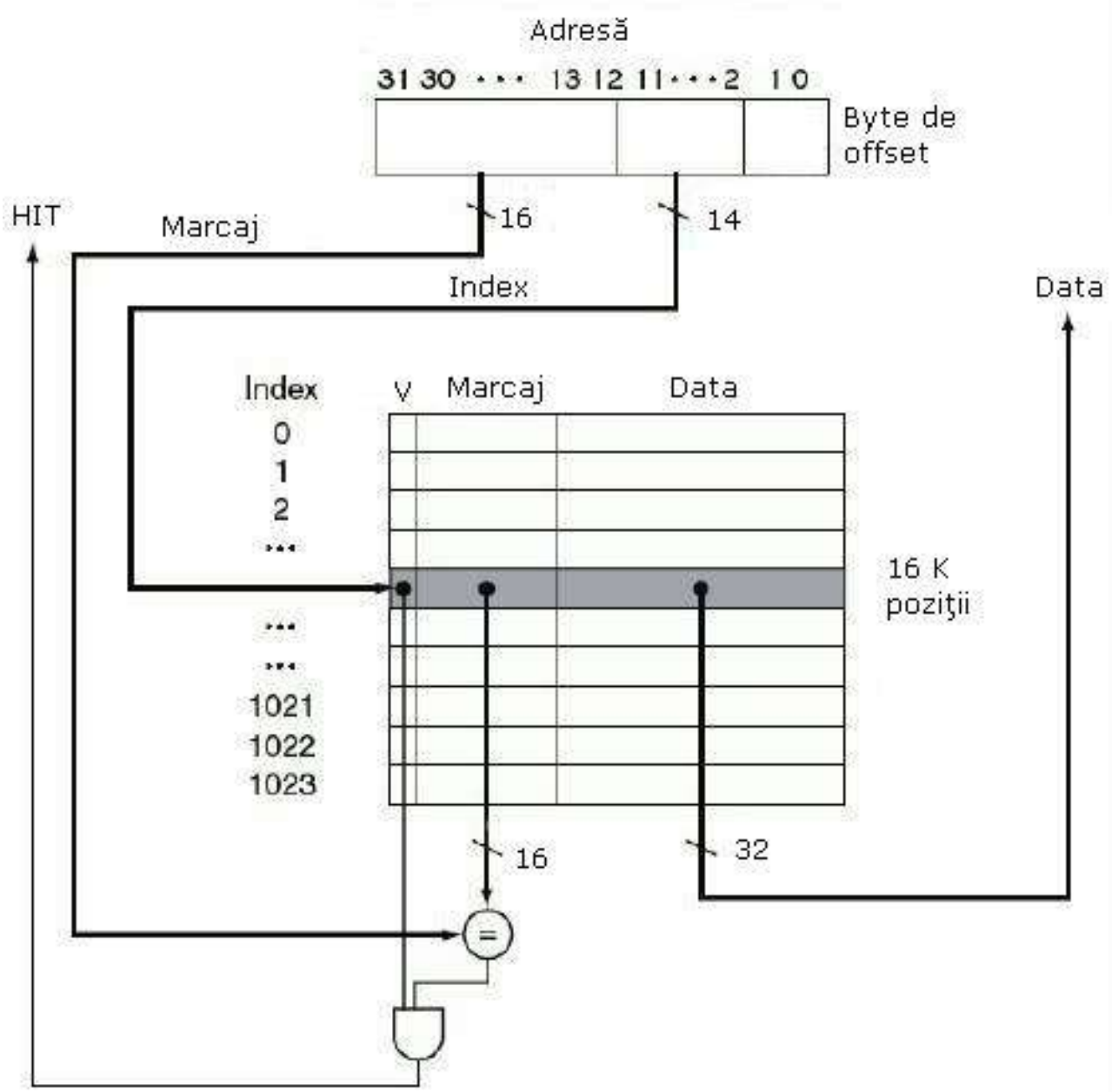
2). se instruieşte memoria principală să execute o citire şi aşteaptă până când acesta termină accesul;

3). se scrie locaţia memoriei cache, punând datele din memorie în porţiunea de date a acestei locaţii, scriind biţii cei mai semnificativi ai adresei (din UAL) în câmpul marcajului şi setând bitul de validare;

4). se reporneşte execuţia instrucţiunii la primul pas, care va reextrage instrucţiunea - de data aceasta se regăseşte în memoria cache.

O metodă de reducere a efectului eşecurilor la memoria cache este folosirea tehnicii **staționare la utilizare**.

EXEMPLU


Adresă
31 30 ... 13 12 11 ... 2 1 0
Byte de offset
16
14
HIT
Marcaj
Index
Data
Index
V
Marcaj
Data
0
1
2
...
...
...
1021
1022
1023
16 K poziţii
16
32
=

CITIREA

1 - se trimite adresa la memoria cache corespunzătoare. Adresa vine fie de la PC (pt instrucţiuni) fie de la UAL (pt date).

2- dacaă HIT cuvântul este disponibil pe liniile de date. Dacă MISS, se trimite adresa la memoria principală. Când memoria transmite datele de la adresa respectivă, acestea sunt scrise în memoria cache.

SCRIEREA

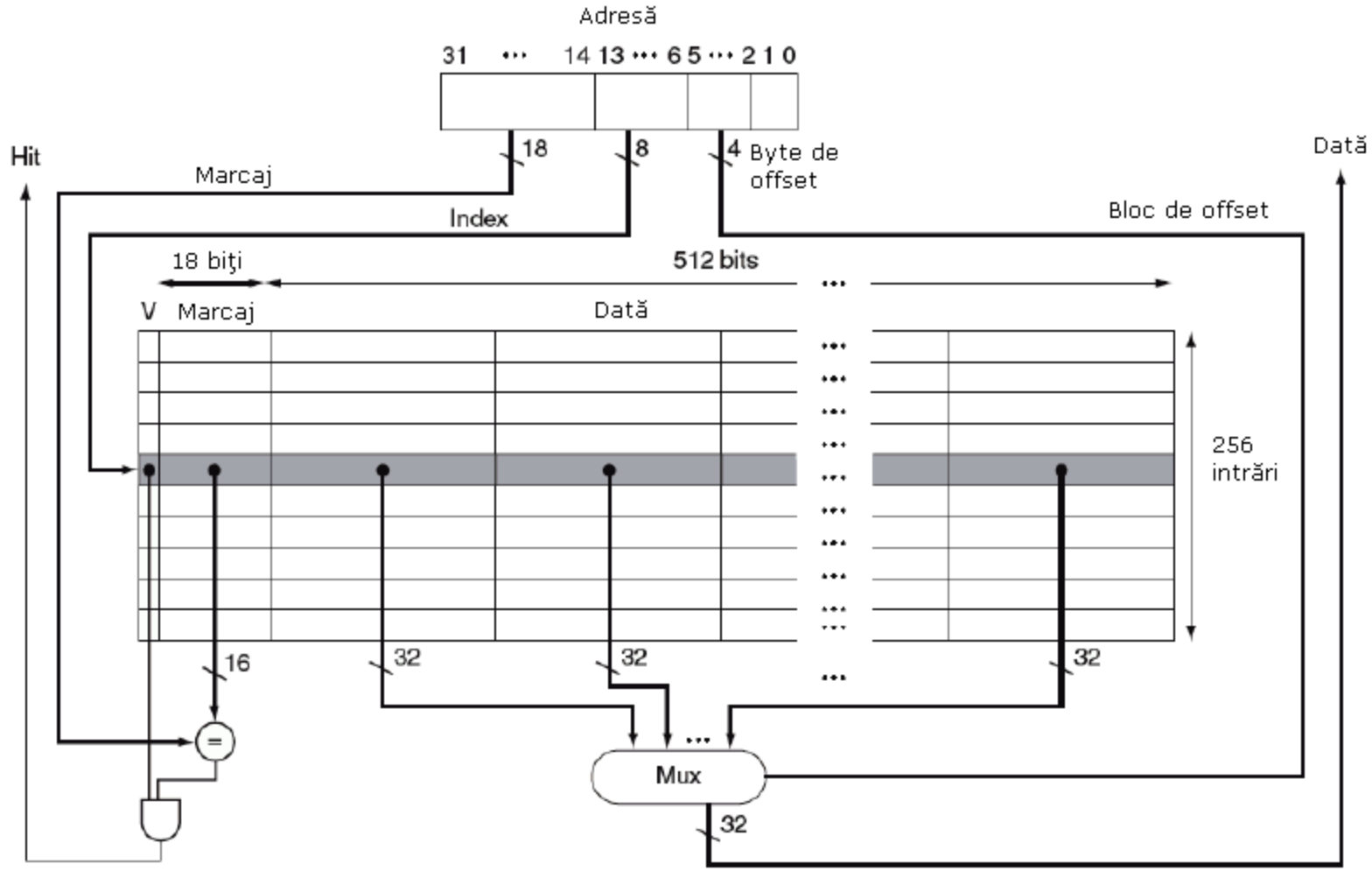
Dacă se scrie doar în memoria cache şi nu şi în memoria principală => memoria principală şi memoria cache sunt inconsistente. Cea mai simplă metodă de evitare este scrierea în ambele memorii => **scriere simultană - write through**.

1. memoria cache este indexată folosind biţii 15-2 ai adresei
2. se scriu biţii 31-16 ai adresei în marcaj, se scrie cuvântul în zona de date şi se setează bitul de validitate
3. se scrie cuvântul în memorie folosind întreaga adresă

SOLUŢIE - folosirea unor memorii tampon - write buffer

O soluţie la metoda **write through** este schema **write back** - scrie la loc

FOLOSIREA LOCALIZĂRII SPAȚIALE



Se doreşte ca blocul memoriei cache să fie mai mare decât lungimea unui cuvânt

Determinarea blocului din memoria cache pentru o anumită adresă

(Adresa blocului) modulo (Numărul de blocuri din memoria cache)

unde

Adresa blocului = adresa cuvântului / numărul de cuvinte din bloc

EXEMPLU

Se consideră o memorie cache cu 64 de blocuri de date, fiecare cu dimensiunea de 16 octeți. Care este numărul blocului corespunzător adresei de octet 1200 ?

Eşecurile şi succesele de scriere

Un bloc de date conţine mai mult de un cuvânt => nu se poate să screim doar marcajele şi datele.

Consideraţii:
1. două adrese de memorie X şi Y au acelaşi bloc corespondent C în memloria cache
2. Blocul are 4 cuvinte şi conţine adresa Y
3. Scriem la adresa X prin simpla suprapunere a datelor şi a marcajului din blocul C

Conform consideraţiilor de mai sus, ce se întamplă după operaţia de scriere ?

Memoria cache
- continuare -

Până acum am folosit doar schema de amplasare a blocurilor din memoria cache denumită corespondență directă.

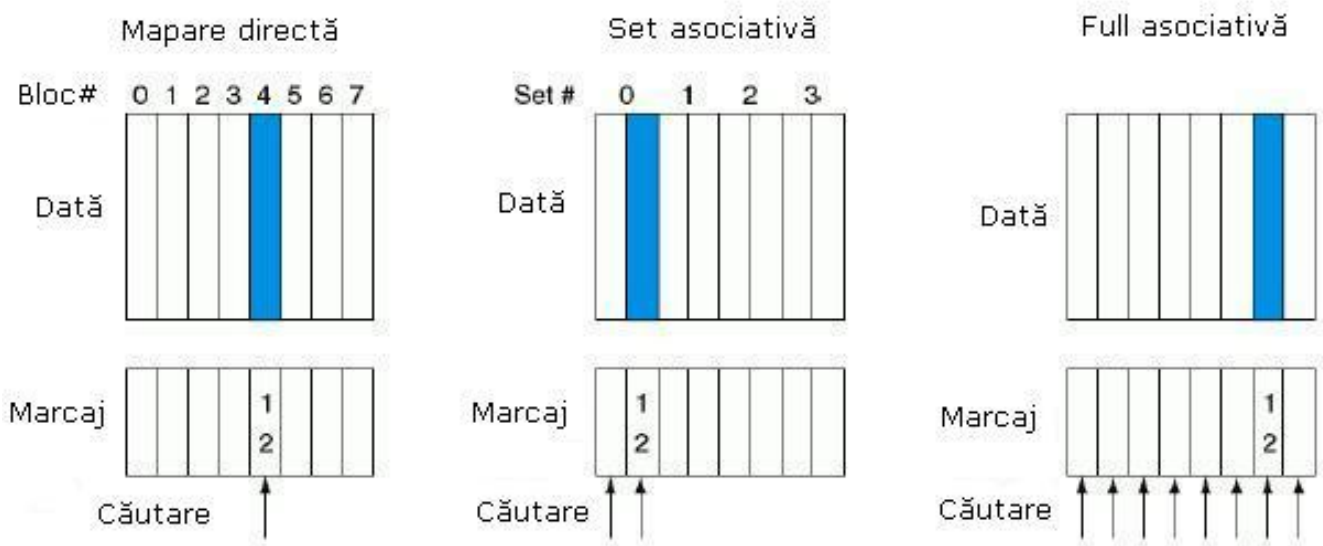
Schema în care un bloc de date poate fi amplasat în orice locație din memoria cache se numeşte schemă cu asociativitate totală.

Regăsirea blocului de date presupune examinarea tuturor locațiilor de memorie cache => paralelizarea căutării

O altă schemă care este între cele două se numeşte schema cu asociativitate parțială.

În aceste tipuri de scheme memoria cache are un număr fix de locații în care se poate amplasa fiecare bloc de date. Deci vom avea un număr de seturi fiecare format din n blocuri de date.

Pentru regăsirea blocului este necesar să parcurgem toate blocurile unui set.



Într-o memorie cache cu asociativitate parțială, setul conținând un anumit bloc de memorie este dat de relația:

(numărul blocului) modulo (numărul de seturi din memoria cache)

OBSERVAȚIE : Creşterea gradului de asociativitate reduce rata de eşec, dar creşte timpul de HIT.

EXEMPLU

Avem 3 memorii cache fiecare având 4 blocuri de câte 1 cuvânt. Cele trei memorii cache sunt cu asociativitate totală, asociativitate cu 2 căi şi corespondență directă.

Să se găsească numărul de eşecuri pentru fiecare dintre cele 3 scheme de amplasare având următoarea secvență de adrese de bloc: 0,8, 0, 6, 8.

SOLUȚIE

Cazul 1 – memoria cache cu corespondență directă

Detectăm blocul din memoria cache corespunzător adreselor date:

Adresa blocului	Blocul memoriei cache
0	0 modulo 4 = 0
6	6 modulo 4 = 2
8	8 modulo 4 = 0

Conținutul memoriei cache după fiecare referință

Adresa blocului de memorie accesat	HIT sau MISS	Blocul 0	Blocul 1	Blocul 2	Blocul 3
0	Miss	Mem(0)			
8	Miss	Mem(8)			
0	Miss	Mem(0)			
6	Miss	Mem(0)		Mem(6)	
8	Miss	Mem(8)		Mem(6)	

Cazul 2. Memoria cache cu asociativitate parțială cu 2 căi conține 2 seturi (indicii fiind 0 şi 1), fiecare având 4 elemente. Vom determina setul corespunzător fiecărei adrese a blocurilor

Adresa blocului	Blocul memoriei cache
0	0 modulo 2 = 0
6	6 modulo 2 = 0
8	8 modulo 2 = 0

Pentru înlocuire vom folosi LRU

Adresa blocului de memorie accesat	HIT sau MISS	Set 0	Set 1	Set 2	Set 3
0	Miss	Mem(0)			
8	Miss	Mem(0)	Mem(8)		
0	HIT	Mem(0)	Mem(8)		
6	Miss	Mem(0)	Mem(6)		
8	Miss	Mem(8)	Mem(6)		

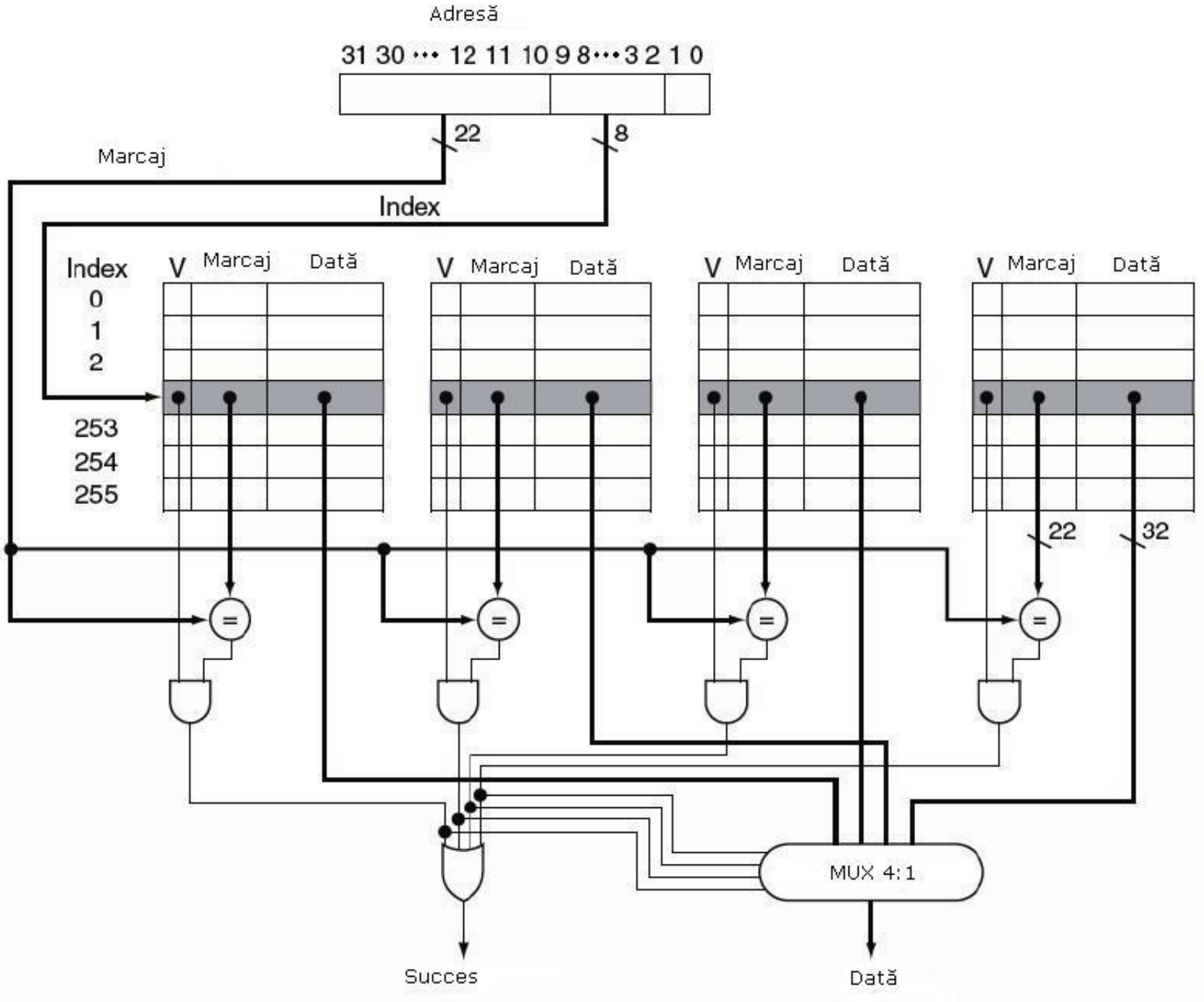
Avem doar 4 eşecuri, deci soluția aceasta este mai bună decât precedenta

Memoria cache cu asociativitate totală

Adresa blocului de memorie accesat	HIT sau MISS	Set 0	Set 1	Set 2	Set 3
0	Miss	Mem(0)			
8	Miss	Mem(0)	Mem(8)		
0	HIT	Mem(0)	Mem(8)		
6	Miss	Mem(0)	Mem(8)	Mem(6)	
8	HIT	Mem(0)	Mem(8)	Mem(6)	

Aceasta este varianta optimă – avem doar 3 eşecuri

Localizarea unui bloc în memoria cache cu asociativitate parțială



Păstrăm dimensiunea memoriei cache constantă şi încercăm să mărim asociativitatea => numărul de blocuri/set va creşte => va creşte numărul de comparații efectuate în paralel.

Creşterea asociativității cu un factor de doi va dubla numărul blocurilor din set şi va înjumătății numărul de seturi => descreşterea dimensiunii indexului cu 1 bit şi o creştere a dimensiunii marcajului cu 1 bit.

Exemplu: Presupunem o memorie cache cu blocuri de 4Kb şi adrese de 32 de biți. Să se găsească numărul total de seturi şi de biți de marcaj pentru memoria cache cu corespondență directă, cu asociativitate parțială cu 2 şi 4 căi şi cu asociativitate totală.

a). Nr. de seturi = nr. de blocuri => log2(4Kb) = 12 biți de index => (32-12)4K=80Kb numărul total al biților din marcaj

b). Cu 2 căi: 2K seturi şi numărul total al biților de marcaj este (32-11)*2*2K = 84Kb
Cu 4 căi: 1K seturi şi numărul total al biților de marcaj este (32-10)*4*1K = 88Kb

c). 1 set cu 4K blocuri iar marcajul are 32 de biți => 32*4K*1 = 128 biți pentru marcaj

MEMORIA VIRTUALĂ

Memoria principală poate acționa ca o memorie cache pentru nivelul de stocare secundar – uzual implementat cu discuri magnetice.

De ce avem nevoie de o memorie virtuală ?

1. Permite folosirea în comun, eficientă şi sigură a memoriei de către mai multe programe

2. Înlăturarea problemelor de programare cauzate de o memorie principală mică

Memoria principală trebuie să conțină doar porțiunile active ale programelor, deci va fi necesar un mecanism de protecție a programelor între ele – trebuie să ne asigurăm că un program va scrie şi va citi doar din memoria principala atribuită lui.

Memoria virtuală implementează translatarea spațiului de adrese al programului în adrese fizice. Acestă translatare asigură unicitatea spațiului de adrese al unui program față de alte programe.

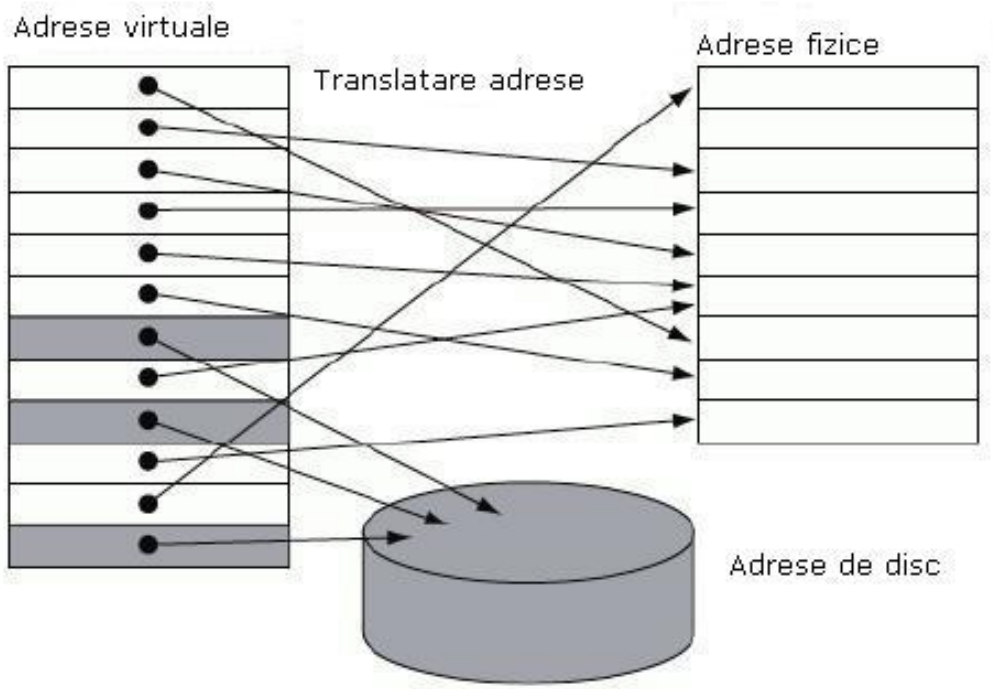
Până la apariția acestui concept, depăşirea dimensiunii de memorie implica intervenția programatorului.

Se împărțea programul în componente şi se trecerea la determinarea excluderilor mutuale.
Suprapunerile erau încărcate sau scoase din memorie în timpul execuției programului.
Apelurile dintre procedurile aflate în diferite module determinau suprapunerea unui modul cu altul.

Un bloc de memorie virtuală este denumit ***pagină***, iar un eşec la accesarea memoriei virtuale se numeşte ***page fault***.

Pentru memoria virtuală vom avea ***adrese virtuale*** ce sunt translatate în ***adrese fizice***.

Exemplu: Adresa virtuală este numele unei cărți iar adresa fizică reprezintă locația cărții în bibliotecă.



Procesorul generează adrese virtuale în timp ce memoria este accesată folosind adrese fizice

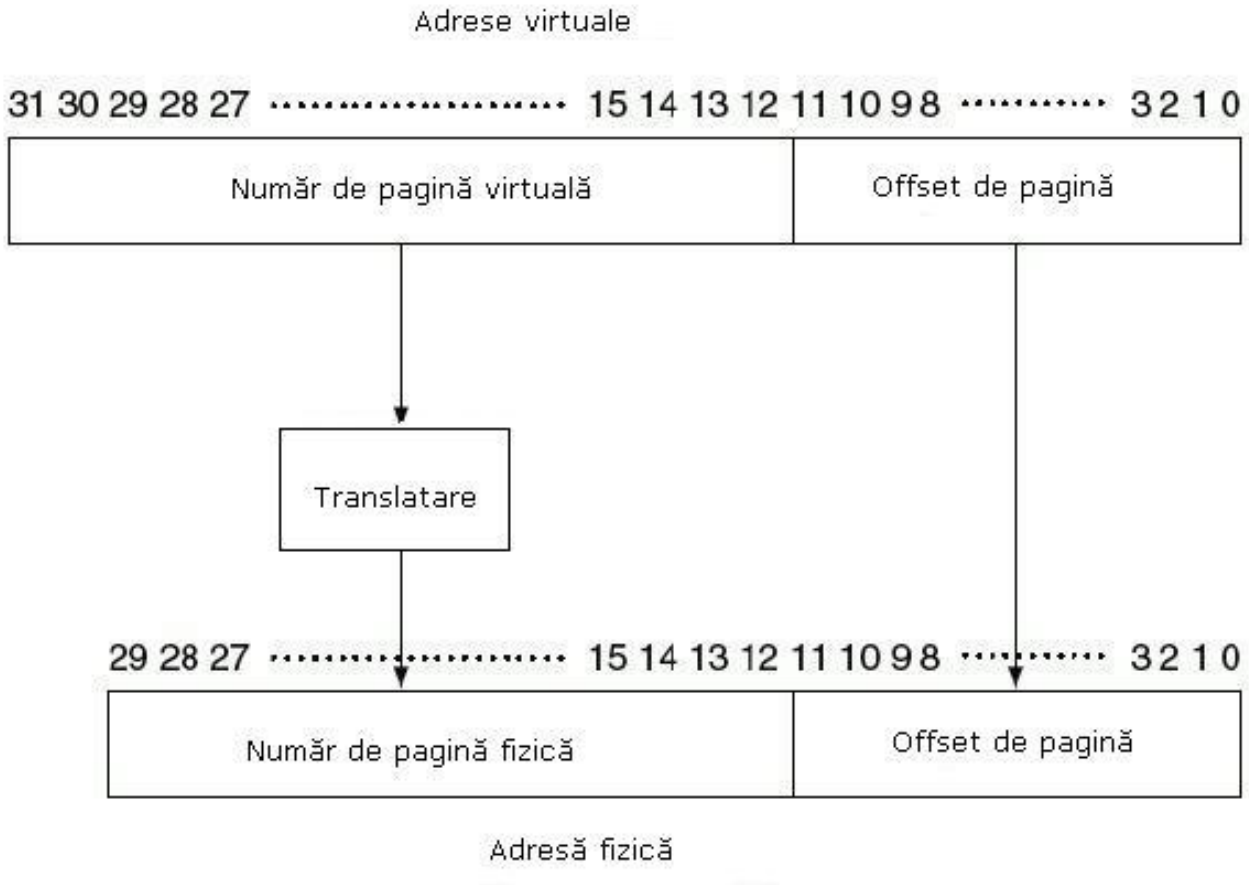
Ambele memorii sunt compuse din pagini (virtuale/fizice) între care există o corespondență de 1:1

Există posibilitatea ca o pagină virtuală să fie prezentă numai pe disc => imposibilitatea de a avea ca şi corespondență o pagină fizică.

O pagină fizică poate fi folosită în comun – două adrese virtuale fac referire la aceeaşi adresă fizică.

Memoria virtuală oferă mecanismul de ***realocare*** – se calculează corespondența dintre adresele virtuale folosite de program şi diferitele adrese fizice , înainte ca adresele fizice să fie folosite de program.

Realocarea se face pe bază de blocuri de dimensiune fixă.



Proiectarea sistemelor de memorie virtuală

Dimensiunea paginilor trebuie să fie mare pentru a amortiza timpul de acces ridicat – 32KB sau 64KB spre exemplu

Amplasarea complet asociativă a paginilor – se reduce complet frecvența de page fault.

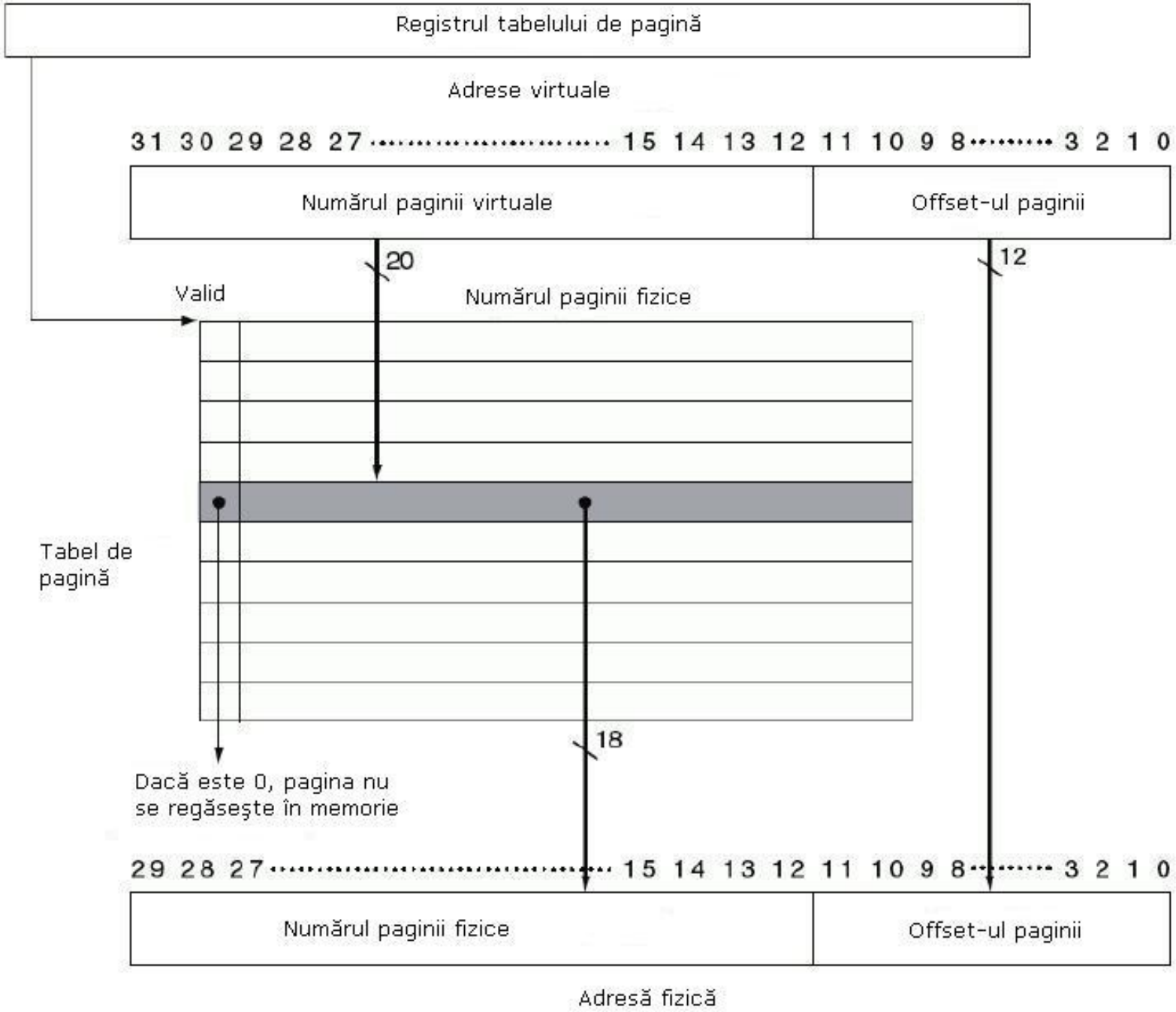
Page fault-urile pot fi tratate prin intermediul software-ului.

În memoria virtuală, paginile sunt localizate prin folosirea unui tabel ce indexează memoria; acestă structură se numeşte ***page table – tabel de pagină***.

Fiecare program are tabelul său de pagini, ce conține corespondența dintre spațiul său de adrese virtuale şi adresele memoriei principale.

Adresa unui tabel este memorată într-un registru ce indică adresa de început a tabelului – ***page table register***.

!!!! Presupunem că tabelul se găseşte într-o regiune fixă şi continuă din memorie


Registrul tabelului de pagină
Adrese virtuale
31 30 29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Numărul paginii virtuale
Offset-ul paginii
20
12
Valid
Numărul paginii fizice
Tabel de pagină
18
Dacă este 0, pagina nu se regăseşte în memorie
29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Numărul paginii fizice
Offset-ul paginii
Adresă fizică

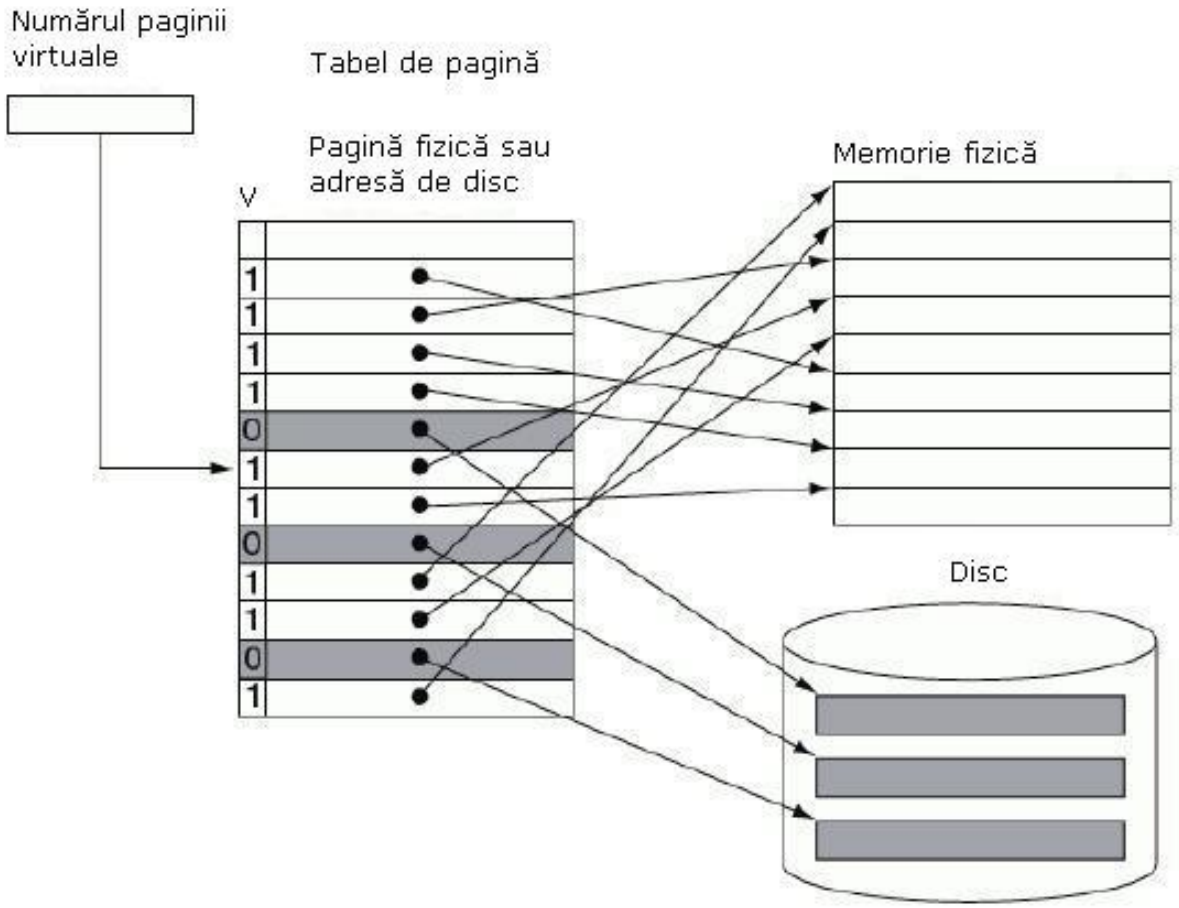
Page faults

În cazul în care bitul de validare este 0 (apare page fault) controlul va fi preluat de SO prin intermediul unui mecanism de tratare a excepțiilor

SO-ul caută pagina în următorul nivel de memorie din ierarhie, adresa virtuală nu poate indica unde se găseşte pagina cerută.

SO-ul creează un spațiu pe disc pentru toate paginile unui proces odată cu crearea acestuia, împreună cu o structură de date pentru înregistrarea locului unde este memorată fiecare pagină virtuală pe disc.

SO-ul creează de asemenea, o structură de date ce ține evidența proceselor şi adreselor virtuale folosite de către fiecare pagină fizică.



Tabelele cu adresele paginilor fizice şi ale paginilor de pe disc vor fi memorate în două structuri de date separate.

Cantitatea de memoria folosită pentru memorarea tabelor de pagini este mare.

Tehnici pentru reducerea volumului de memorie ocupat de tabelele de pagini şi minimizarea memoriei principale alocate acestora

1. Utilizarea unui registru de limitare ce constrânge dimensiunea tabelului de pagini pentru un proces dat.

2. Majoritatea limbajelor necesită 2 porțiuni expandabile – una ce conține stiva şi cealaltă ce conține zona de acumulare (HEAP).

Dezavantaj – nu funcționează bine atunci când spațiul de adrese este folosit într-un mod discontinuu.

3. Aplicarea unei funcții de căutare – HASHING – pentru adresa virtuală, astfel încât structura de date ce conține tabelul de pagini să aibă o dimensiune egală doar cu numărul de pagini fizice existente în memoria principală.

4. Paginarea tabelor de pagini.

5. Utilizarea mai multor niveluri de tabele de pagini.

Memoria virtuală
- continuare -

Putem folosi scheme de memorii cache cu scrieri simultane doar dacă implementăm bufer-e de memorie (diferența dintre timpul de acces al memoriei cache şi cel al memoriei principale este de ordinul zecilor de cicluri)

Sistemele de memorie virtuală trebuie să implementeze scrierea la loc (copy back) – se execută scrieri individuale în pagina de memorie şi se copiază această pagină înapoi pe disc, atunci când este înlocuită în memorie.

Va trebui să determinăm dacă o pagină de memorie necesită transferul înapoi atunci când ea este înlocuită în memorie.

Pentru aceasta adăugăm un bit de scriere ‘’dirty bit’’. Acest bit inițial 0 va deveni 1 în momentul în care se scrie prima dată în această pagină.

Mărirea vitezei de translatare a adresei

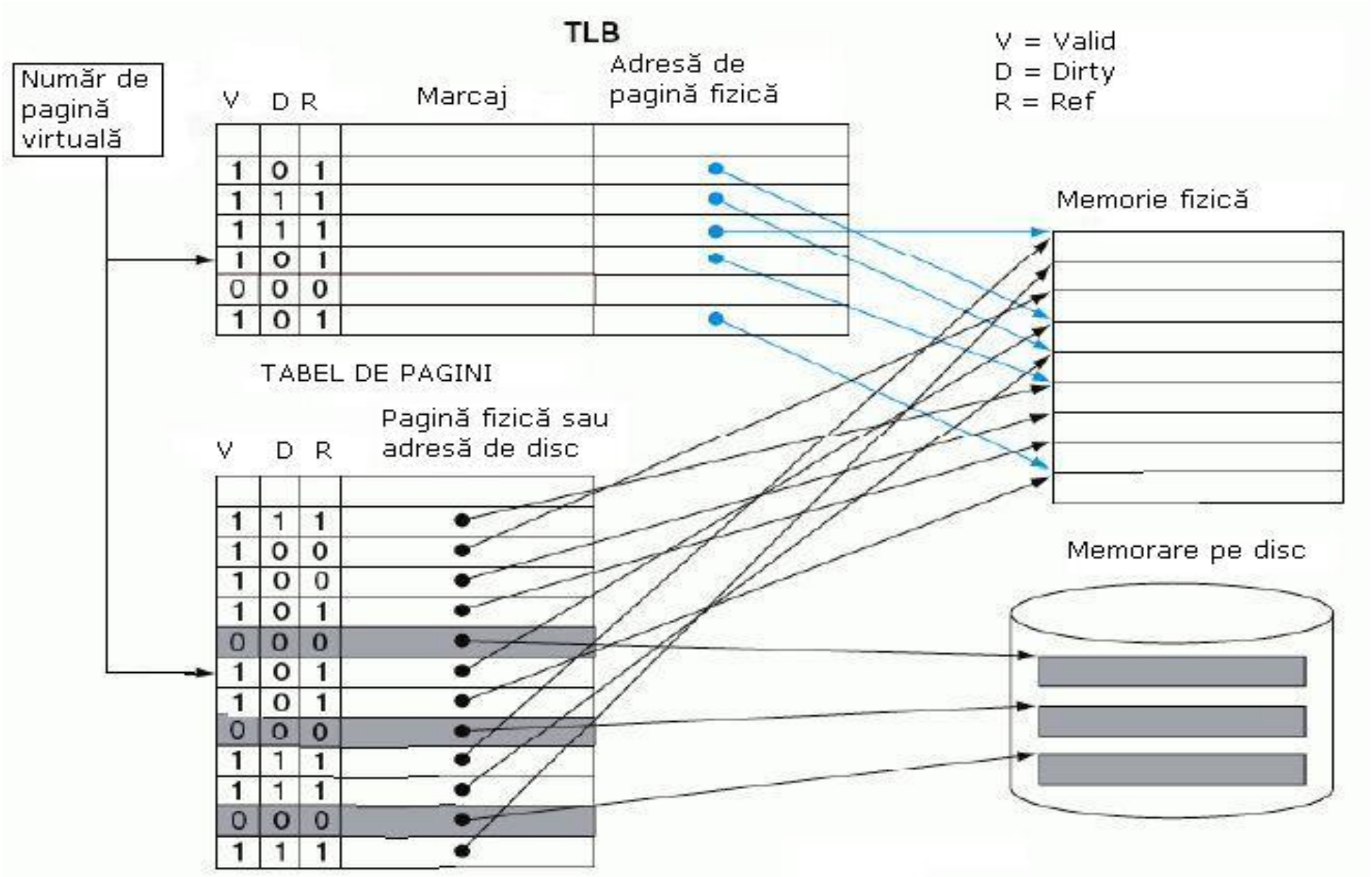
Unde sunt păstrate tabelele de pagini ?

Ce accese la memorie sunt necesare pentru un program ?

Ce principiu putem utiliza pentru mărirea performanței ?

Când este folosită o translatare pentru un număr de pagină virtuală, aceasta va fi probabil necesară din nou în viitorul apropiat. De ce ?

Din aceste considerente, maşinile moderne includ o memorie cache specială care menține evidența translatărilor recente numită **Translation Lookaside Buffer – memorie tampon de translatare cu căutare laterală - TLB**



TLB acționează ca şi o memorie cache a tabelului de pagini doar pentru pozițiile acestuia ce au corespondent în paginile fizice

La fiecare referință se va căuta numărul de pagină virtuală din TLB.

Daca avem HIT, numărul paginii fizice este utilizat pentru formarea adresei, iar bitul de referință corespunzător va fi setat. Bitul de scriere va fi setat şi în cazul în care procesorul execută o scriere

Dacă avem MISS trebuie determinată cauza: greşeală de pagină sau eşec în TLB.

Cazul 1 – eşec în TLB, deci pagina este în memorie => că translatarea lipseşte. UCP-ul va trata eroarea prin încărcarea translatării din tabelul de pagini în TLB şi reluarea referinței.

Cazul 2 – greşeală de pagină, UCP-ul va invoca sistemul de operare folosind o excepție.

Eşecurile în TLB vor fi mult mai frecvente decât greşelile de pagină. **De ce ?**

Eşecurile în TLB vor fi tratate software sau hardware. Ambele metode oferă performanțe similare.

Integrarea memoriilor

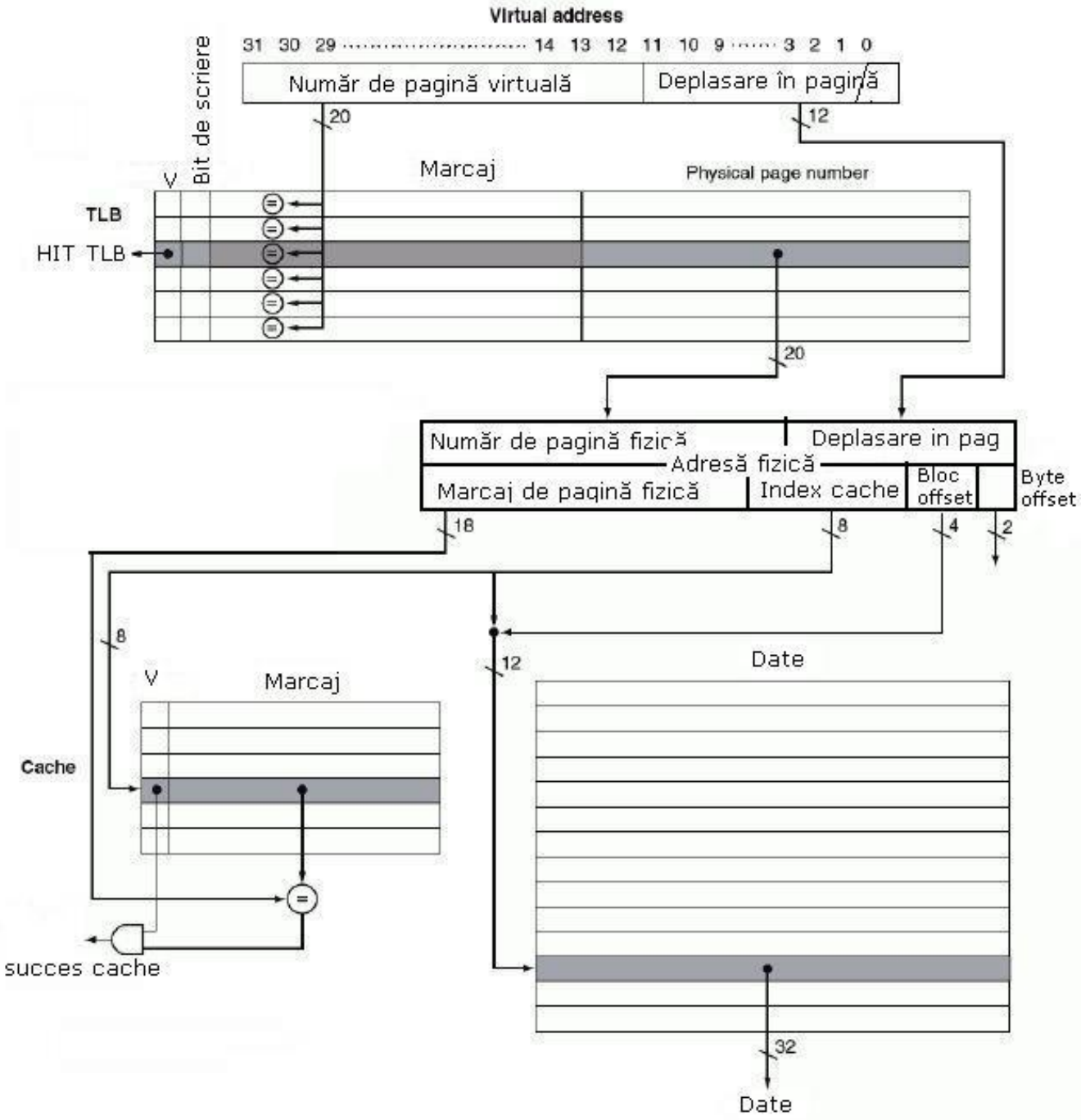
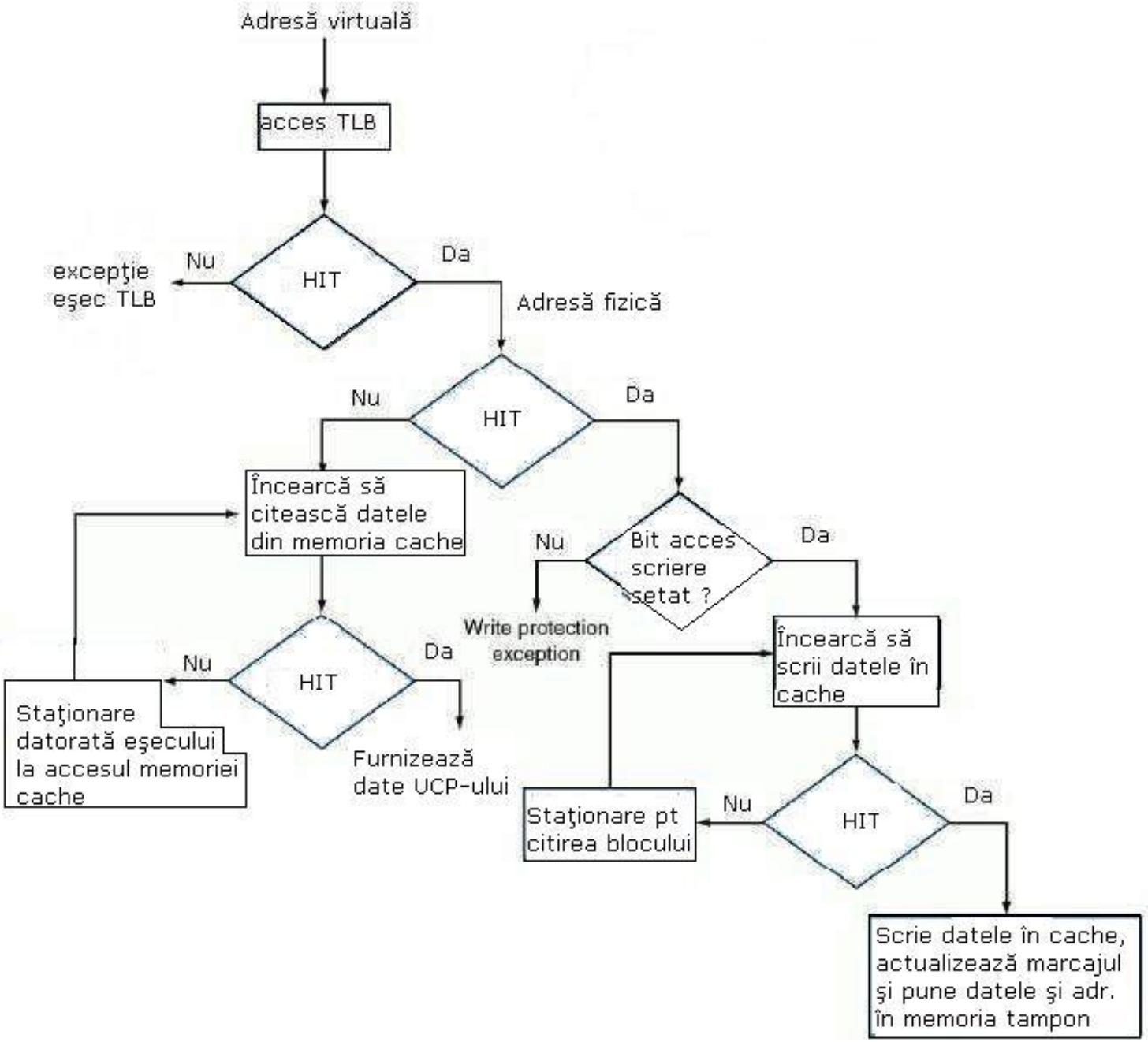


Fig 1.

În timp ce memoria cache are corespondență directă, TLB-ul are asociativitate totală. Implementarea asociativității totale pentru TLB necesită ca fiecare marcaj din TLB şi fie comparat cu valoarea indexului T deoarece poziția căutată poate fi oriunde în TLB.

Dacă bitul de validare al poziției corecte este setat, accesul la TLB este un succes şi numărul de pagină împreună cu deplasarea acesteia formează indexul ce este folosit pentru accesul la memoria cache.

Citirea sau scrierea cu un TLB şi o memorie cache – DEC Station 3100



Considerăm ierarhia de memorie din figura 1 ce include:

- un TLB
- o memorie cache

O referință la memorie poate avea următoarele 3 tipuri de eşecuri:

- un eşec al memoriei cache
- un eşec al TLB-ului
- greşeală de pagină

Memorie cache	TLB	Memorie virtuală	Eveniment posibil ? În ce condiții ?
MISS	HIT	HIT	
HIT	MISS	HIT	
MISS	MISS	HIT	
MISS	MISS	MISS	
MISS	HIT	MISS	
HIT	HIT	MISS	
HIT	MISS	MISS	

Exemplul prezentat presupune că adresele de memorie sunt translatate în adrese fizice înainte ca memoria cache să fie accesată => memoria cache este indexată şi marcată fizic.

Timpul de acces la memorie (în cazul HIT) = timpul de acces TLB + timp acces memoria cache – accesele pot fi executate în pipe.

O alternativă la această soluție este aceea ca UCP-ul să indexeze memoria cache cu o adresă parțial/complet virtuală – **MEMORIE CACHE ADRESATĂ VIRTUAL.**

În astfel de implementări, memoria cache este indexată şi marcată virtual.

Hardware-ul pentru translatarea adresei (exp. TLB-ul) nu este folosit în timpul accesului normal la memoria cache De ce ?

În cazul unor astfel de implementări, când paginile sunt folosite în comun de către programe, există posibilitatea de suprapunere - **ALIASING**

Protecția folosind memoria virtuală

Funcția cea mai importantă a unei memorii virtuale – ***folosirea de către mai multe procese în comun a memoriei principale***.

Mecanismul de protecție trebuie să asigure:
• un proces nu poate să scrie în spațiul de adrese al altui proces sau în sistemul de operare
• nu trebuie să fie posibilă situația în care un proces citeşte datele altui proces

Pentru asigurarea acestor cerințe hardware-ul trebuie să ofere:

• trebuie să suporte cel puțin două moduri ce indică dacă procesul executat este un proces al utilizatorului sau un proces al sistemului de operare (kernel process sau supervisor process)

• trebuie să furnizeze o porțiune a stării UCP-ului pe care să o poată citi, dar nu şi scrie un proces al utilizatorului – sistemul de operare foloseşte instrucțiuni speciale oferite doar în modul supervizor

• trebuie să ofere mecanisme prin care UCP poate trece din modul utilizator în cel supervizor şi invers

Tratarea greşelilor de pagină şi a eşecurilor în TLB

Un eşec TLB apare atunci când nici o poziție în TLB nu corespunde unei anumite adrese virtuale. Putem avea următoarele posibilități:

1. Pagina este prezentă în memorie şi trebuie creată poziția lipsă din TLB
2. Pagina nu este prezentă în memorie şi trebuie ca SO-ul să să prelucreze această situație

   Cum stabilim care situație este situația curentă ?

Un eşec TLB poate fi tratat software sau hardware – va trebui o secvență scurtă de operațiuni pentru a copia o poziție validă din tabelul de pagini în TLB.

Excepția cauzată de greşeala de pagină trebuie declarată până la sfârşitul aceluiaşi ciclu de ceas în care se face accesul la memorie.

CONCLUZII

Memoria virtuală reprezintă numele unui nivel din ierarhia de memorie ce administrează caching-ul dintre memoria principală şi unitatea de disc.

Permite unui singur program să îşi extindă spațiul de adrese în afara limitelor memoriei principale.

Memoria virtuală permite folosirea în comun a memoriei principale de către mai multe procese simultan active oferind mecanisme de protecție a memoriei.

Tehnicile existente pentru administrarea ierarhiei de memorie între memoria principală şi unitatea de disc sunt:

1. Folosirea de blocuri de date de dimensiuni mari (pagini) pentru folosirea principiului localizării spațiale
2. Corespondența dintre adresele virtuale şi cele fizice – implementată printr-un tabel de pagini – este total asociativă => o pagină virtuală poate fi plasată oriunde în memoria principală.
3. SO-ul foloseşte LRU sau bitul de referință pentru alegerea paginii care trebuie înlocuită

Mecanismul memoriei virtuale oferă translatarea din adrese virtuale în adrese fizice. Acestă translatare permite folosirea în comun – în mod partajat – a memoriei principale.

TLB-ul acționează ca o memorie cache pentru translatările din tabelul de pagini. Fiecare adresă este translatată dintr-una virtuală în una fizică folosind translatările din TLB.

UNDE POATE FI AMPLASAT UN BLOC DE DATE ?

Amplasarea blocului la nivelul superior al ierarhiei. Avantajul creşterii gradului de asociativitate este acela al scăderii ratei de eşec.

Performanța creşte prea puțin în cazul măririi dimensiunii memoriei cache deoarece rata de eşec globală a unei memorii cache de dimensiuni mari este mai redusă.

Dezavantajul schemelor cu asociativitate este dat de costurile crescute şi de timpul de acces mai mare.

CUM SE POATE REGĂSI UN BLOC DE DATE ?

Implementarea unui grad înalt de asociativitate în memoriile cache nu este o soluție optimă datorită costurilor comparatoarelor care cresc în timp ce rata îmbunătățirile ratei de eşec sunt mici.

În sistemele de memorie virtuală se păstrează un tabel separat de corespondență pentru indexarea memoriei. Alegerea asociativității totale şi a acestui tabel sunt justificate de 4 factori. Care sunt aceeia ?

Schemele de amplasare cu asociativitate parțială sunt folosite pentru memoriile cache şi TLB-uri, unde accesul combină indexarea şi căutarea într-un set redus de locații.

CE BLOC DE DATE TREBUIE ÎNLOCUIT ÎN CAZ DE MISS PENTRU MEMORIA CACHE ?

În cazul în care memoria este cu asociativitate totală, toate blocurile pot fi înlocuite. Dacă memoria cache este parțial asociativă, trebuie ales între blocurile din set. Dacă avem memorie cu corespondență directă, atunci avem o singură posibilitate de înlocuire.

Strategiile fundamentale de înlocuire sunt: aleatorie şi LRU. LRU nu se implementează în ierarhii cu mai mult de 2-4 grade de asociativitate. Pentru restul de ierarhii LRU este doar aproximat.

În memoriile cache algoritmul de înlocuire este codificat în hardware. Odată cu creşterea dimensiunii memoriei cache rata de eşec pentru ambele startegii de înlocuire scade şi diferența devine foarte mică.

CE SE ÎNTÂMPLĂ ÎN CAZUL SCRIERILOR ?

Există 2 metode de bază:

Scrierea simultană – informația este scrisă atât în blocul de date din memoria cache cât şi în blocul de date din nivelul inferior al ierarhiei.

Scrierea la loc – informația este scrisă doar în blocul din memoria cache. Blocul modificat este scris la nivelul ierarhic inferior doar atunci când este înlocuit.

AVANTAJE pentru scrierea la loc:

1. Cuvintele individuale pot fi scrise de procesor la viteza cu care sunt acceptate de către memoria cache şi nu de către memoria principală.

2. Scrierile multiple în interiorul blocului necesită o singură scriere în nivelul inferior al ierarhiei de memorie

3. Pentru scrierea blocurilor la nivelul inferior se foloseşte un transfer de bandă mare.

AVANTAJE pentru scrierea simultană:

1. **Eşecurile sunt mai simple şi au un cost mai scăzut, deoarece nu necesită scrierea unui bloc la loc în nivelul inferior de memorie**

2. Este un sistem mai uşor de implementat în hardware

I / O

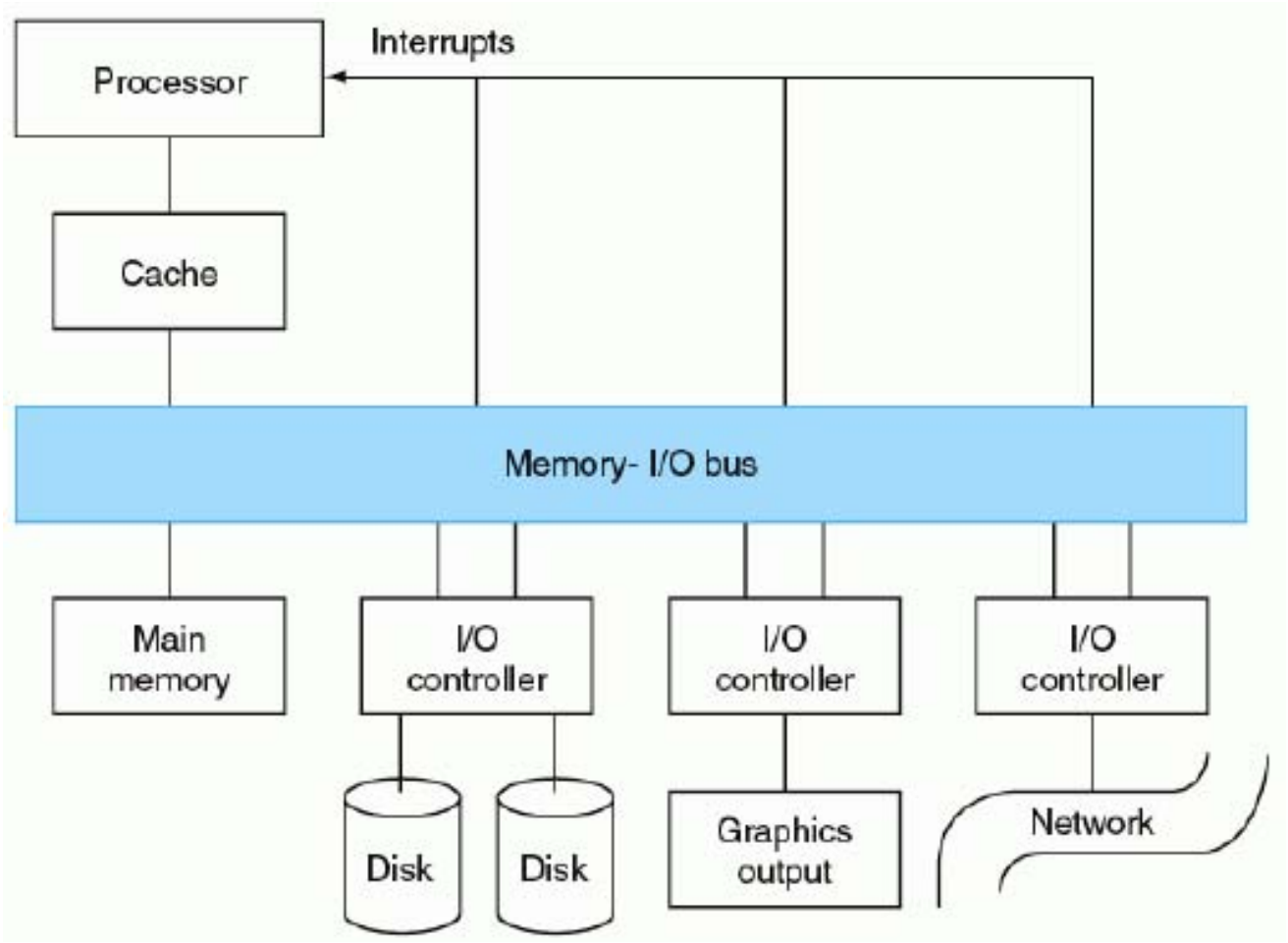
Actualmente există o mare varietate de dispozitive de I/O. Organizarea acestor dispozitive se poate face având în vedere următoarele caracteristici:

1. COMPORTAREA – input (citire o singură dată); output (scriere odată); storage (citire și rescriere)

2. Partenerul – om sau dispozitiv electronic

3. Rata datelor – valoarea maximă cu care datele pot fi transferate între dispozitivul de I/O și memoria principală/procesor

Exp: Tastatura este un dispozitiv de intrare utilizat de către om cu o rată a datelor de peste 10 bytes/sec



Conexiunile între dispozitivele de I/O, procesor și memorie sunt denumite magistrale.
Comunicația între dispozitive și procesor presupune utilizarea întreruperilor precum și folosirea unor protocoale de comunicație.

Performanța I/O depinde de lățimea de bandă existentă între dispozitive. Lățimea de bandă poate fi măsurată prin 2 metode:

1. Cât de multe date pot fi mutate prin sistem într-o anumită perioadă de timp

2. Câte operații de I/O pot fi efectuate într-o unitate de timp

Exemple: În cazul aplicațiilor multimedia lățimea de bandă este folosită în determinarea performanței; în cazul procesărilor unui număr foarte mare de accese al unui dispozitiv I/O, numărul de operații I/O efectuate în unitatea de timp va fi factorul cheie în determinarea performanței.

În cazul calculatoarelor și al laptop-urilor, timpul de răspuns este considerat factorul cheie în determinarea performanței

În cazul dispozitivelor embedded ne interesează durata fiecărui task și numărul de task-uri ce pot fi procesate într/o secundă.

Discurile magnetice – platane rotitoare acoperite cu o suprafață magnetică ce utilizează mișcarea capetelor de citire/scriere pentru accesul la disk.

Este nonvolatil – datele rămân și după întreruperea alimentării cu energie a dispozitivului.

1-4 platane, fiecare având 2 suprafețe ce pot fi scrise;

Stiva de platane este rotită cu o viteză de 5400 – 15000 RPM.

Diametrul platanelor este de la 1 inch la peste 3,5 inch

Fiecare suprafață de disk este împărțită în cercuri concentrice denumite *piste.* Ele sunt în număr de 10000 – 50000 pe o singură suprafață.

Fiecare pistă este împărțită în sectoare (100-500). Fiecare sector are o dimensiune tipică de 512bytes.

Secvența de scriere este următoarea:

număr sector
gap
informația pentru sector + codul de corecție eroare
gap
numărul umătorului sector

Inițial toate pistele aveau același număr de sectoare – s-a introdus ZBR (zone bit record)

Capetele de citire/scriere sunt conectate împreună => mișcarea se face în conjuncție. Fiecare cap este peste aceeași pistă indiferent de suprafață => cilindru.

Accesarea unei date presupune:

1. Poziționarea capetelor deasupra pistei dorite – seek – seek time

2. Se așteaptă până când capetele ajung deasupra sectorului dorit – delay sau rotational delay

Exp:

Average rotational latency = $\frac{0,5}{5400 RPM} = \frac{0,5 \quad rotatii}{5400\ RPM\ /\ (60\ \frac{s}{m})} = 0,0056\ s = 5,6\ ms$

Average rotational latency = $\frac{0,5}{15000 RPM} = \frac{0,5 \quad rotatii}{5400\ RPM\ /\ (60\ \frac{s}{m})} = 0,0020\ s = 2,0\ ms$

3. Timpul de transfer – timpul necesar transferării unui bloc de biți.

Timpul de trasfer = f(dimensiune sector, viteza de rotație, densitatea înregistrărilor a unei piste)

Controller-ul de disk – are rol de control al discului precum și de control al transferului dintre disk și memorie.

La timpul de acces la disk se include și timpul necesar operării controller-ului de disk

Exp: Să se determine timpul mediu de citire/scriere al unui sector de 512bytes pentru un disk care are o viteză de rotație de 10000 RPM. Se cunosc:

1. *Timpul mediu de poziționare dat = 6 ms*
2. *Rata de transfer = 50 MB/s*
3. *Overhead-ul controller-ului este de 0,2 ms*
4. *Presupunem că discul este idle => nu există timp de așteptare*

Soluție

Timpul mediu de acces al discului = timpul mediu de poziționare + întârzierea medie + timpul de transfer + overhead-ul controller-ului

$$6{,}0\ ms + \frac{0{,}5\ rot}{10000\ RPM} + \frac{0{,}5\ KB}{50\ MB/s} + 0{,}2\ ms = 6{,}0 + 3{,}0 + 0{,}01 + 0{,}2 = 9{,}2\ ms$$

Dacă timpul mediu de poziționare măsurat este 25% din timpul mediu dat, atunci avem: 1,5 ms + 3,0 ms + 0,01 ms + 0,2 ms = 4,7 ms

RAID – Redundant Arrays of Inexpensive Disks

O organizare de disk-uri care utilizează o matrice de disk-uri mici (ca și capicate) și ieftine pentru creșterea performanțelor și a siguranței în utilizare.

Ideea a fost de înlocuire a discurilor mari cu disk-uri mici. Disk-urile mici sunt mult mai eficiente per gigabyte decât disk-urile mari (evident ne referim la cantitatea de date stocată pe un astfel de disk)

RAID 0 – nu avem redundanță – striping

Presupune răspândirea datelor pe mai multe disk-uri => acces automat la mai multe disk-uri simultan. Din punct de vedere al utilizatorului există doar un singur disk, ceea ce simplifică managementul informației.

Performanță mare pentru accese la informație de dimensiune mare (sisteme de editare video) deoarece mai multe disk-uri funcționează ca unul singur.

RAID 1 – toleranță la defecte – mirroring sau shadowing

Este modalitatea aleasă atunci când toleranța la defecte este un punct critic. Numărul de hardisk-uri utilizat este dublu față de RAID 0.

Când o dată este scrisă pe un disk, automat datele sunt scrise pe un alt disk redundant => întotdeauna vom avea 2 copii ale informației. Dacă un disk va prezenta un defect la un moment dat, atunci informația va fi citită de pe discul oglindă. CEA MAI SCUMPĂ SOLȚIE RAID.

RAID 2 – detectare și corectare erori

Împrumută tehnicile de detecție și corecție a erorilor folosite în cazul memoriilor

RAID 3 – grup de protecție – bit-interleaved parity

Costul unei disponibilități mărite a datelor poate fi redus la 1/N, unde N este numărul de disk-uri care fac parte dintr-un grup de protecție.

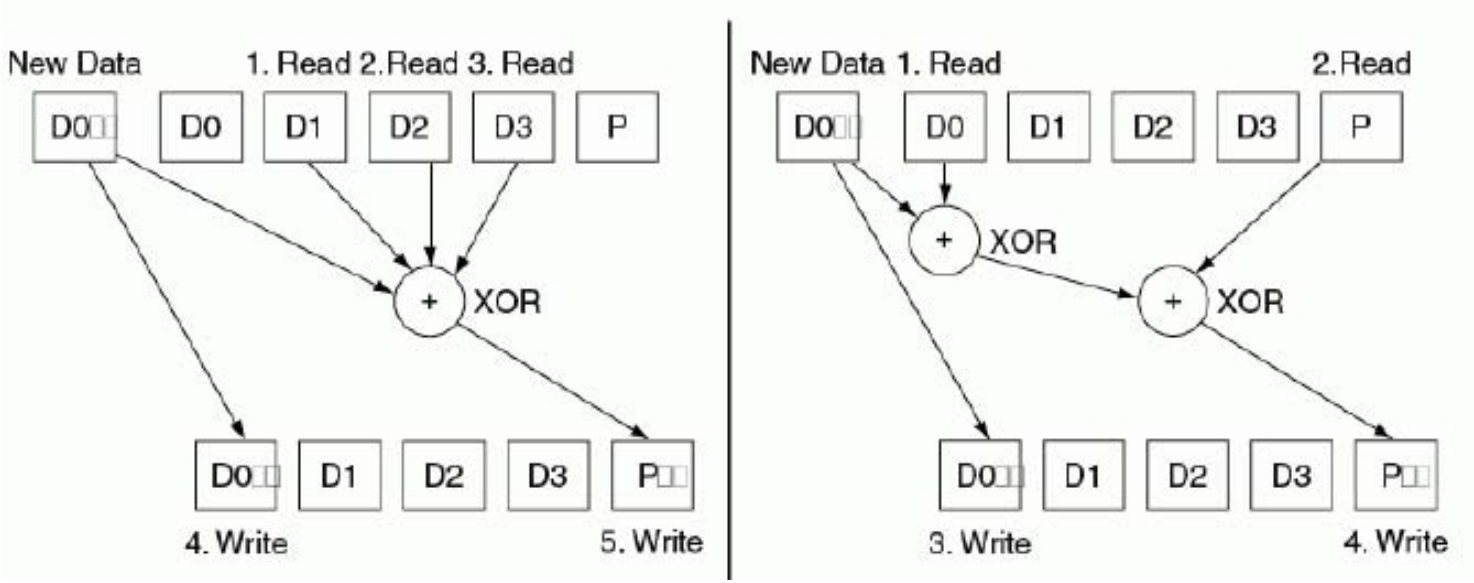
Decât să adăugăm disk-uri, mai simplu ar fi să adăugăm informație redundantă pentru restaurarea informației pierdute în caz de crash.

Citirile/scrierile se fac pe toate disk-urile din grup, dar vom avea 1 extra disk pentru menținerea informației de verificare în caz de crash. Schema folosită este determinarea parității informației.

RAID 3 este foarte folosit în cazul aplicațiilor care utilizează seturi de date foarte mari – multimedia sau cod științific

RAID 4 – block-interleaved parity

Similar cu RAID 3, doar că utilizează un acces al datelor diferit. Paritatea este memorată ca blocuri și asociată cu un set de blocuri de date.



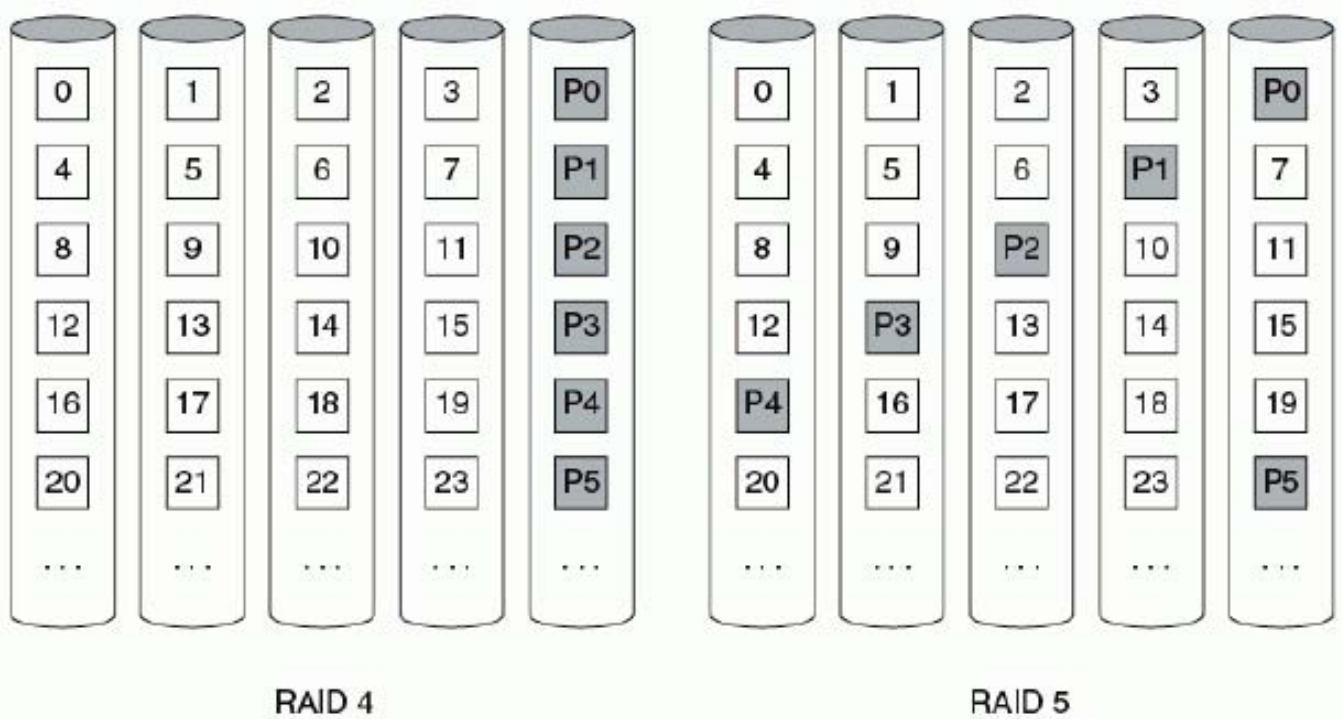
RAID 3 vs RAID 4

Sunt optimizate scrierile mici, deci vom avea un număr redus de accese la disk precum și un număr mic de disk-uri ocupate.

RAID 5 – distributed block-interleaved parity

Dezavantajul lui RAID 4 este faptul că paritatea disk-ului trebuie recalculată la fiecare scriere.

O soluție ar fi să distribuim această informație pe toate discurile astfel încât să nu mai avem un singur bottleneck.



Prin această soluție, anumite scrieri mici pot fi executate în paralel.

RAID 6 – P + Q redundancy

Folosit în cazul în care o singură corecție nu este suficientă. Putem generaliza paritatea pentru a avea o nouă calulație pentru date și o nouă informație pentru verificarea discului.

Acest block secundar este folosit pentru recuperarea datelor în caz de eșec multiplu. Overhead-ul este dublu față de RAID 5.

Alte metode folosite în practică

1. HOT SWAPPING – replasarea unei componente hardware cât timp sistemul este în stare de funcționare

2. STANDBY SPARES – Cuplarea unor resurse hardware noi imediat ce o resursă hardware este defectă

MAGISTRALE - BUSES

O magistrală conține un set de linii de control și un set de linii de date.

Liniile de control sunt utilizate pentru semnale de tip cerere și confirmare și ele indică tipul informației existentă pe liniile de date.

Liile de date sunt folosite la transportarea informației între sursă și destinație. Acestă informație poate să conțină comenzi complexe, date și adrese.

Bus transaction – o secvență de operații care include o cerere și poate include un răspuns. O tranzacție este inițiată de un singur request și poate avea mai multe operații individuale de magistrală.

Processor-memory bus

Backplane Bus - o magistrală care este proiectată pentru a permite coexistarea pe o singură magistrală a urmaătoarelor componente: procesor, echipamente I/O, memoria

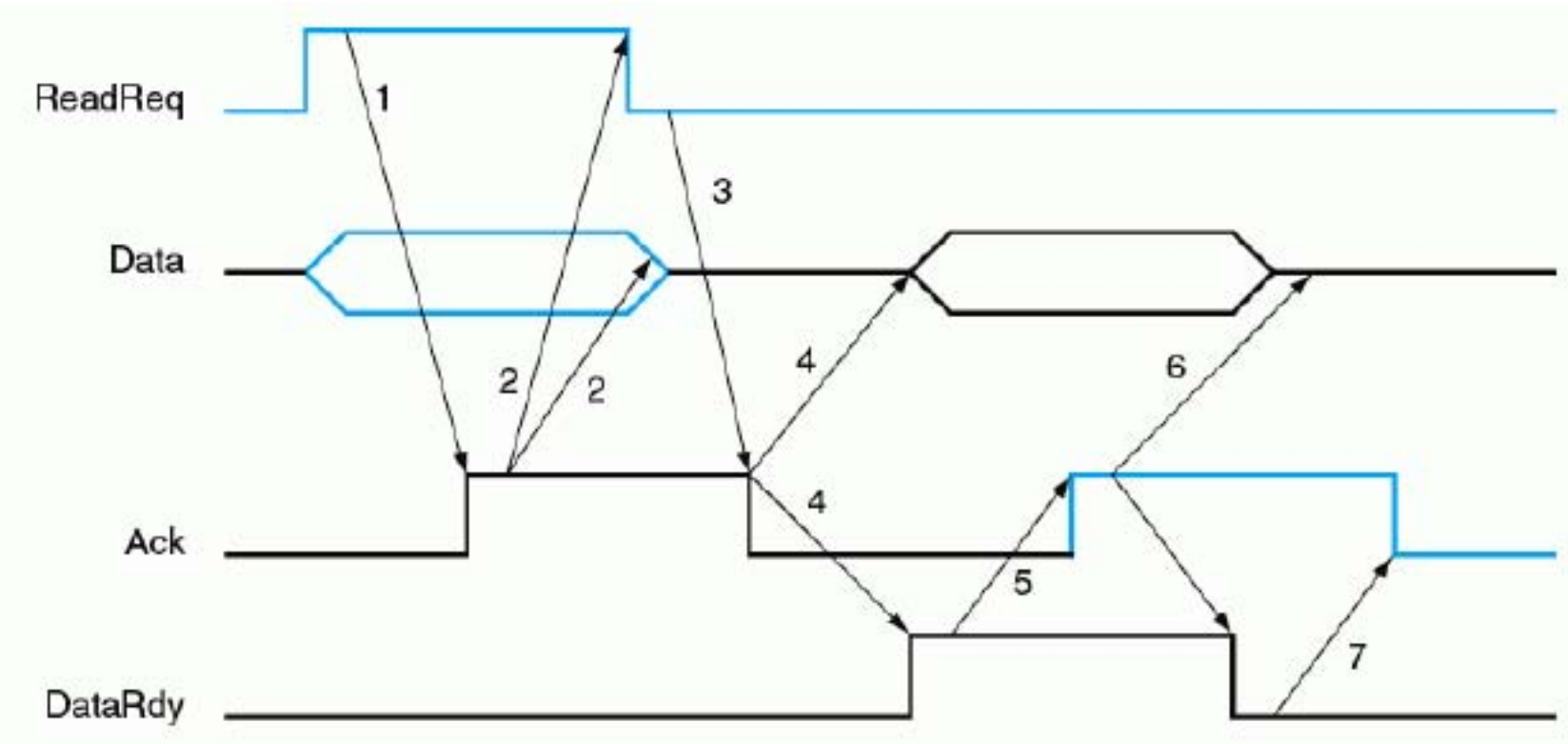
Magistrală sincronă – O magistrală care include ceasul în liniile de control și un protocol fix pentru comunicarea relativă la frontul de ceas.

Magistrală asincronă – O magistrală care utilizează protocolul HANDSHAKING pentru coordonarea utilizării în locul ceasului. Este folosită ca o punte de legătură între dispozitive care operează la viteze diferite

Bus type	I/O	I/O
Basic data bus width (signals)	4	2
Clocking	asynchronous	asynchronous
Theoretical peak bandwidth	50 MB/sec (Firewire 400) or 100 MB/sec (Firewire 800)	0.2 MB/sec (low speed), 1.5 MB/sec (full speed), or 60 MB/sec (high speed)
Hot plugable	yes	yes
Maximum number of devices	63	127
Maximum bus length (copper wire)	4.5 meters	5 meters
Standard name	IEEE 1394, 1394b	USE Implementors Forum

Firewire (1394) USB 2.0

Protocolul - HANDSHAKING



Split Transaction Protocol – Un protocol în care magistrala este eliberată pe durata unei transacții de magistrală cât timp cel care a lansta cererea este în așteptare pentru datele ce vor fi transmise.

Studiu de caz – PENTIUM 4