Arhitectura sistemelor de calcul

- Prelegerea 11 -

3-DS (Procesoare)

Ruxandra F. Olimid

Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Cuprins

3-DS (Procesoare)

- Introducem încă un *ciclu* la sistemele cu 2 cicluri (2-DS) şi obţinem sistemele *3-DS* care introduc un grad sporit de autonomie

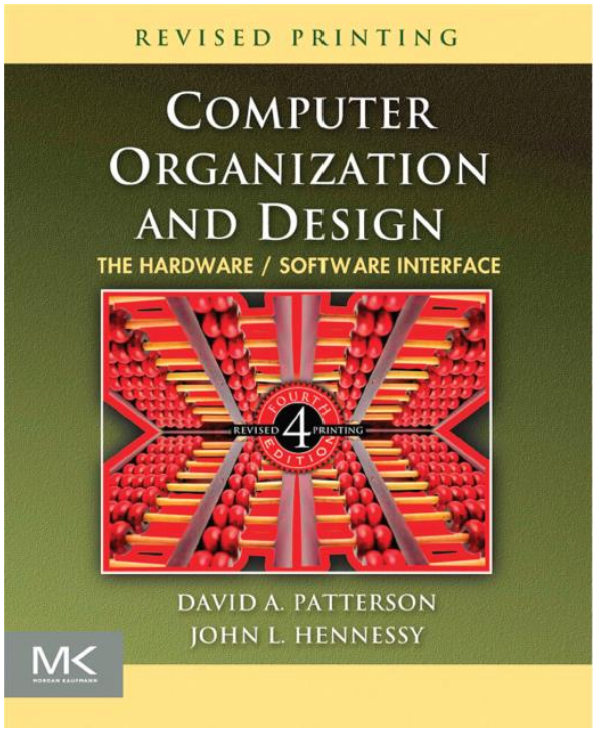
- Închiderea celui de-al treilea ciclu peste un 2-DS se poate realiza:
  - ✓ printr-un ciclu care implică pur circuite combinaţionale (0-DS)
  - ✓ printr-un ciclu care implică memorie (1-DS)
  - ✓ printr-un ciclu care implică un automat (2-DS)

- Un exemplu cunoscut de *3-DS* este *procesorul*

- Vom studia procesorul MIPS din clasa RISC, prezentat pe larg în [COD]

Implementare (I)

- Vom studia procesorul MIPS din clasa RISC, prezentat pe larg în cartea suport a cursului:



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

Implementare (I)

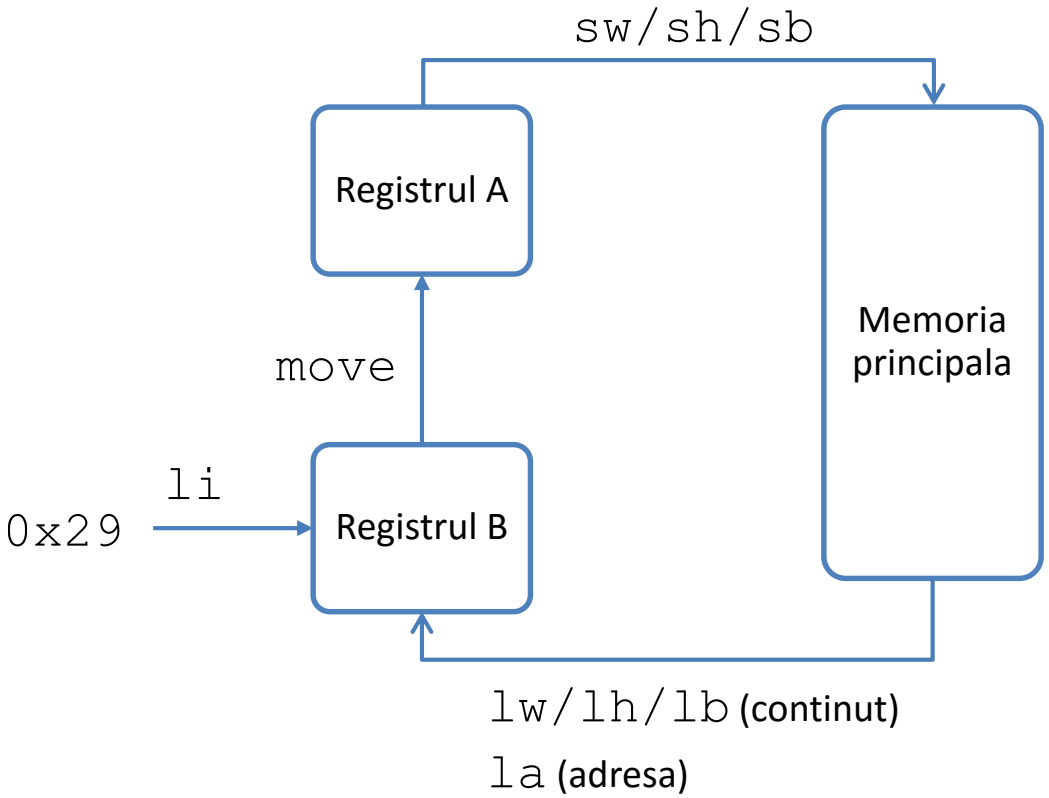
- Restricţionăm studiul la o variantă simplificată a MIPS care implementează o parte din setul total de instrucţiuni:
  - ✓ intrucţiuni de lucru cu memoria: load (`lw`) şi store (`sw`)
  - ✓ intrucţiuni aritmetice şi logice: `add`, `sub`, `and`, `or`, `slt`
  - ✓ intrucţiuni condiţionale: `beq`, `j`

- *Întrebare:* Ce înseamnă *setul de instrucţiuni*?

- *Răspuns:* Setul de instrucţiuni reprezintă totalitatea comenzilor întelese de processor. Am studiat la laborator clasificarea procesoarelor ca *CISC* (*C*omplex *I*nstruction *S*et *C*omputer) sau *RISC* (*R*educed *I*nstruction *S*et *C*omputer), în funcţie de complexitatea instrucţiunilor.

Implementare (I)

- *Întrebare:* Ce înseamnă `lw`? Dar `sw`?
- *Răspuns:* `lw` = load word; `sw` = store word



Implementare (I)

- Instrucţiunile considerate pentru implementare acoperă cele 3 formate posibile:
  - ✓ format *R* (`add`, `sub`, `and`, `or`, `slt`)
  - ✓ format *I* (`lw`, `sw`, `beq`)
  - ✓ format *J* (`j`)

Reprezentarea instrucţiunilor în calculator

- Instrucţiunile sunt stocate în binar; fizic, sunt secvenţe de semnale înalte (1) sau joase (0)

- O instrucţiune MIPS32 ocupă **32 de biţi** (**4 locaţii de memorie** sau **1 word**)

- Regiştrii se reprezintă în instrucţiune pe **5 biţi** (sunt 32 de regiştrii generali, numerotaţi de la 0 la 31)

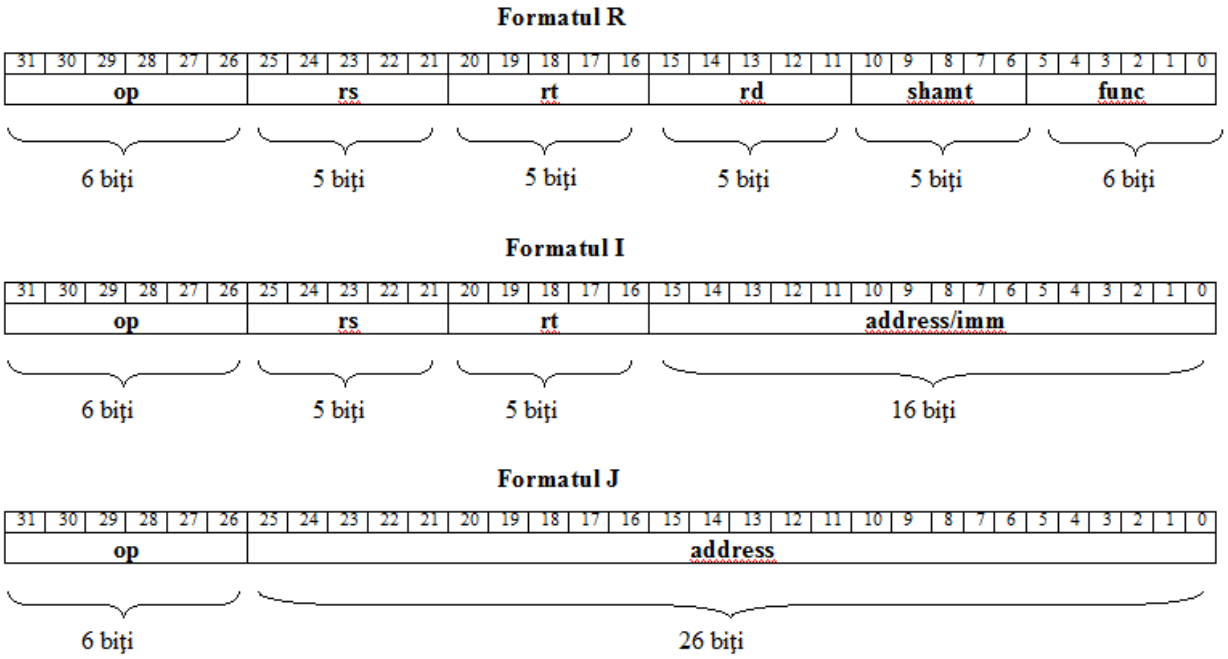
- *Exemplu:* `$s1 = $17;  $s2 = $18; $s3 = $19`, deci:

  `add $s1, $s2, $s3` , devine `add $17, $18, $19`

Reprezentarea instrucţiunilor în calculator

- *Formatul instrucţiunii* este modul de reprezentare al instrucţiunii prin spargerea în câmpuri cu semnificaţie pentru procesor

- Pentru MIPS32, instrucţiunile respectă unul dintre următoarele 3 formate :

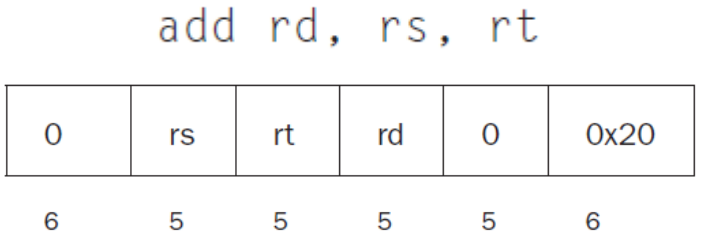


Reprezentarea instrucţiunilor în calculator

- *op* = **operaţia de bază** (opcod)
  - În cazul instrucţiunilor în format R, op este întotdeauna 000000
  - În cazul instrucţiunilor în format J, op este întotdeauna de forma 00001x, cu x cifră binară
  - În cazul instrucţiunilor în format I, op diferă, însă nu este niciodată de forma 000000, 00001x sau 0100xx, cu x cifră binară
- *rs* = **registru sursă** – registrul care conţine primul argument
- *rt* = **registru sursă** – registrul care conţine al doilea argument (în cazul instrucţiunilor în format R) sau **registrul destinaţie** (în cazul instrucţiunilor în format I)
- *rd* = **registru destinaţie** – registrul în care se stochează rezultatul obţinut în urma operaţiei
- *shamt* = **shift amount** – folosit la operaţiile de deplasare (shiftare)
- *func* = **funcţie** – combinată cu op indică operaţia/funcţia care se aplică
- *address* = **adresă**
- *imm* = **valoare imediată**

Reprezentarea instrucţiunilor în calculator

add $s1, $s2, $s3



[COD]

Instrucţiune: `add $17, $18, $19`

Hex: `0x02538820`

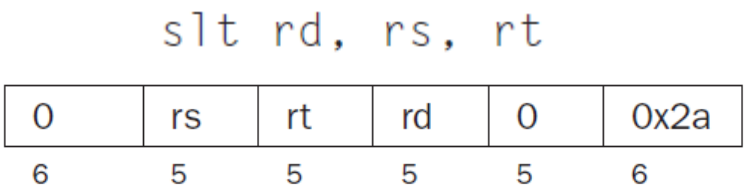
Binar: `0000 0010 0101 0011 1000 1000 0010 0000`

Format **R**: **000000 10010 10011 10001 00000 100000**

`op = 000000`	(format R)
`rs = 10010`	($18 = $s2)
`rt = 10011`	($19 = $s3)
`rd = 10001`	($17 = $s1)
`shamt = 00000`	(nu se face shiftare)
`func = 100000`	(adunare)

Reprezentarea instrucţiunilor în calculator

slt $s5, $s6, $s7



[COD]

Instrucţiune: `slt $21, $22, $23`

Hex: `0x02d7a82a`

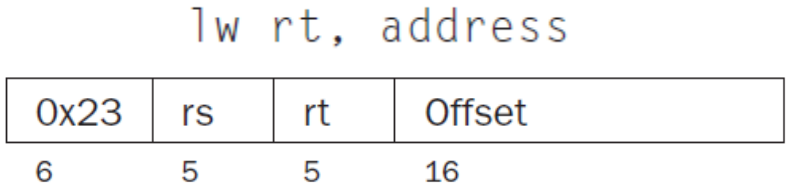
Binar: `0000 0010 1101 0111 1010 1000 0010 1010`

Format **R**: `000000 10110 10111 10101 00000 101010`

`op = 000000`	(format R)
`rs = 10110`	($22 = $s6)
`rt = 10111`	($23 = $s7)
`rd = 10101`	($21 = $s5)
`shamt = 00000`	(nu se face shiftare)
`func = 101010`	(set on less than)

Reprezentarea instrucţiunilor în calculator

lw $t1, 4($t0)



Instrucţiune: `lw $9, 4($8)` [COD]

Hex: `0x8d090004`

Binar: `1000 1101 0000 1001 0000 0000 0000 0100`

Format **I**: `100011 01000 01001 0000000000000100`

`op = 100011` (format I, load word)

`rs = 01000` ($8 = $t0)

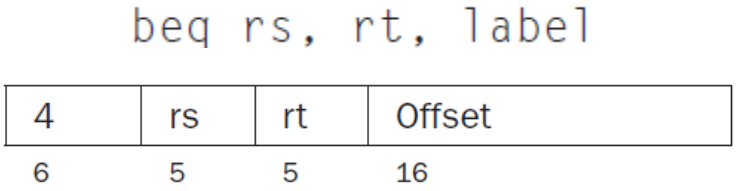
`rt = 01001` ($9 = $t1)

`imm = 0000000000000100` (offset faţa de adresa din $t1)

Reprezentarea instrucţiunilor în calculator

beq $t1, $zero, sfarsit

[sfarsit este o eticheta din program]



[COD]

Instrucţiune: `beq $9, $0, 28`

Hex: `0x11200007`

Binar: `0001 0001 0010 0000 0000 0000 0000 0111`

Format **I**: `000100 01001 00000 0000000000000111`

`op = 000100` (format I, branch on equal)

`rs = 01001` ($9 = $t1)

`rt = 00000` ($0 = $zero)

`imm = 0000000000000111` (salt cu 28 locatii de memorie, adica 7 instructiuni)

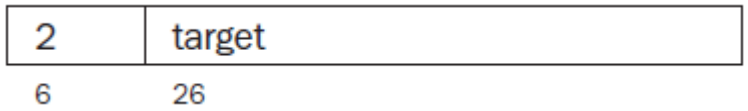
[Vom analiza ulterior cum se interpreteaza de fapt campul offset / imm pe implementarea procesorului]

Reprezentarea instrucţiunilor în calculator

j sfarsit

[sfarsit este o eticheta din program]

j target



[COD]

Instrucţiune: `j 0x0040002c`

Hex: `0x0810000b`

Binar: `0000 1000 0001 0000 0000 0000 0000 1011`

Format **J**: `000010 00000100000000000000001011`

`op = 000010` (format J)

`address = 00000100000000000000001011`

(salt la adresa indicata)

[Nota: singurele 2 instructiuni de tip J sunt j (op=000010) si jal (op = 000011)]

[Vom analiza ulterior cum se interpreteaza de fapt campul address pe implementarea procesorului]

Reprezentarea instrucţiunilor în calculator

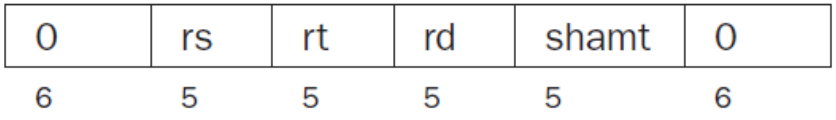
sll $t1, $t2, 2

- *Întrebare:* Ce ştiţi despre această instructiune? Completaţi pe modelul exemplelor anterioare informaţiile pe care le puteţi deduce.

Reprezentarea instrucţiunilor în calculator

sll $t1, $t2, 2

sll rd, rt, shamt



[COD]

Instrucţiune: `sll $9, $10, 2`

Hex: `0x000a4880`

Binar: `0000 0000 0000 1010 0100 1000 1000 0000`

Format **R**: `000000 00000 01010 01001 00010 000000`

`op = 000000`	(format R)
`rs = 00000`	($0 = $zero)
`rt = 01010`	($10 = $t2)
`rd = 01001`	($9 = $t1)
`shamt = 00010`	(se face shiftare cu 2)
`func = 000000`	(shift left logical)

Implementare (I)

- *Întrebare:* Unde se încarcă intrucţiunile unui program?
- *Răspuns:* În zona de memorie de instrucţiuni (text segment)

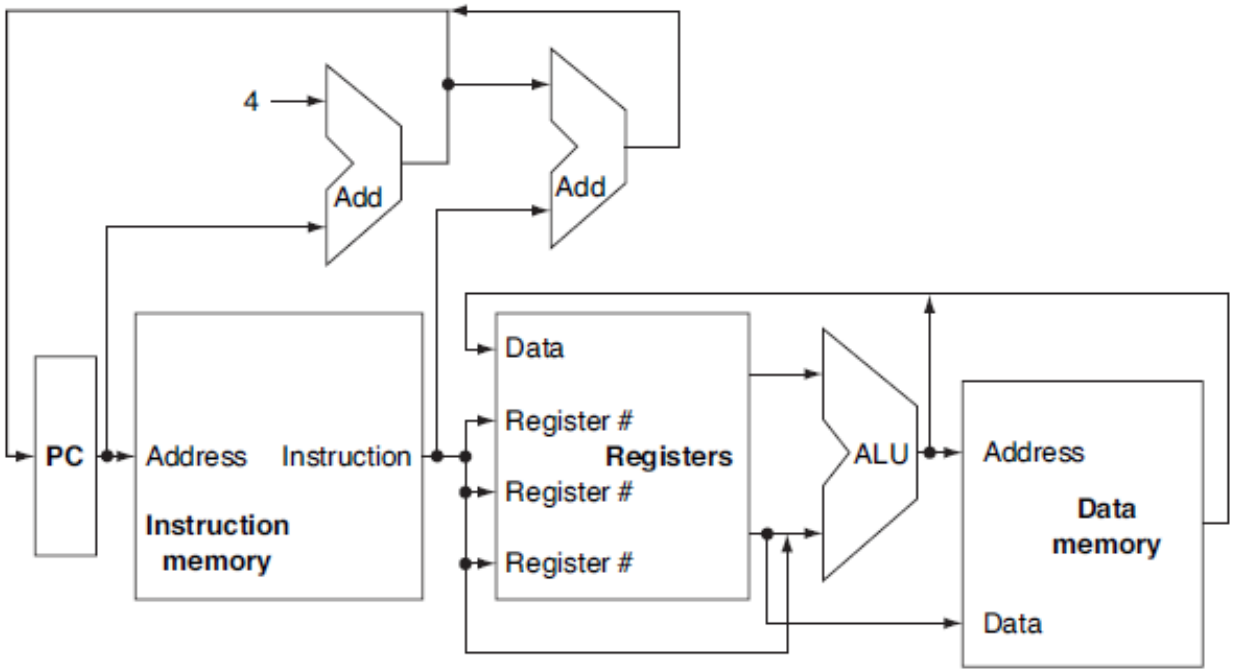
```
.data                    #  declaratii date (memoria de date)
        …
.text                    # identifică porţiuni cu instrucţiuni
                         # (memoria de instructiuni)
        …
main:                    # eticheta marcând punctul de start
        …

li $v0,10                # terminarea executiei programului
syscall
```

Implementare (I)

- *Întrebare:* Cum se decide care este următoarea instrucţiune care va fi executată?
- *Răspuns:* Dacă nu există salt, instrucţiunile se execută secvenţial. Registrul special *PC* (*P*rogram *C*ounter) conţine adresa următoarei instrucţiuni care va fi executată.
- *Întrebare:* Cu cât se incrementează PC pentru a trece la instrucţiunea următoare (fără să se considere salt)?
- *Răspuns:* Cu 4 pentru că o instructiune MIPS32 ocupă 32 de biţi, adică 4 locaţii de memorie.

Implementare (I)

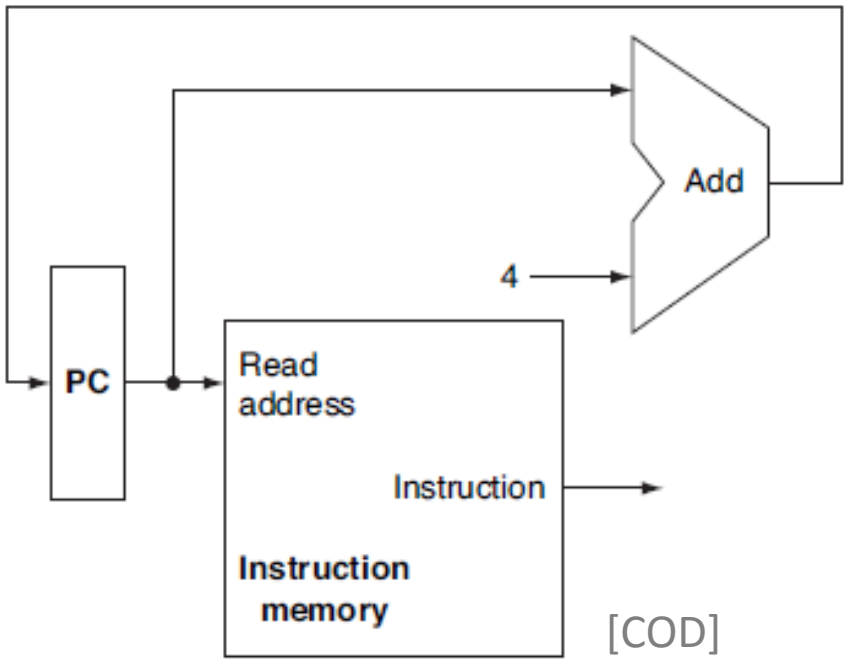
- O primă privire simplă de ansamblu (fără menţionarea semnalelor de control şi toate funcţionalităţile):



[COD]

Extragerea instrucţiunii

- Pentru **extragerea unui instrucţiuni şi trecerea la instrucţiunea următoare** sunt necesare:
  - ✓ **Memoria de instrucţiuni**: locul unde se păstrează instrucţiunile
  - ✓ **PC** (Program Counter): rergistrul special care indică instrucţiunea următoare care va fi executată
  - ✓ **Adder**: dacă nu se realizează salt, trecerea la următoarea instrucţiune se face prin adunare cu 4

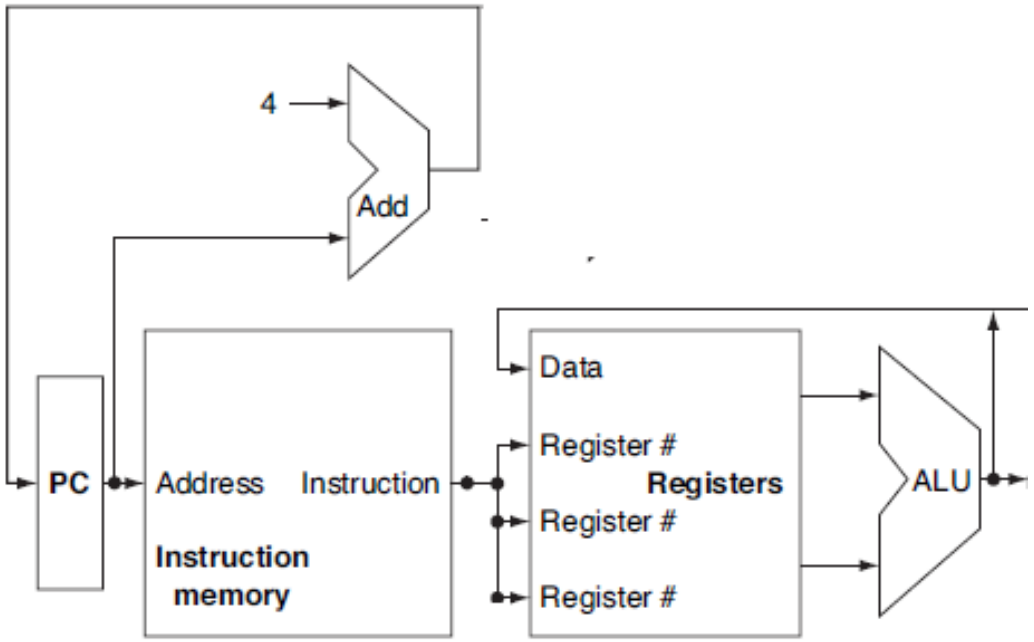


[COD]

Instrucţiuni aritmetice / logice

- Pentru **instrucţiunile aritmetice şi logice** (în format **R**) sunt necesare:
  - ✓ **File Register:** pentru cei 2 regiştrii operanzi şi 1 registru destinaţie
  - ✓ **ALU:** pentru realizarea operaţiei aritmetice sau logice şi obţinerea rezultatului

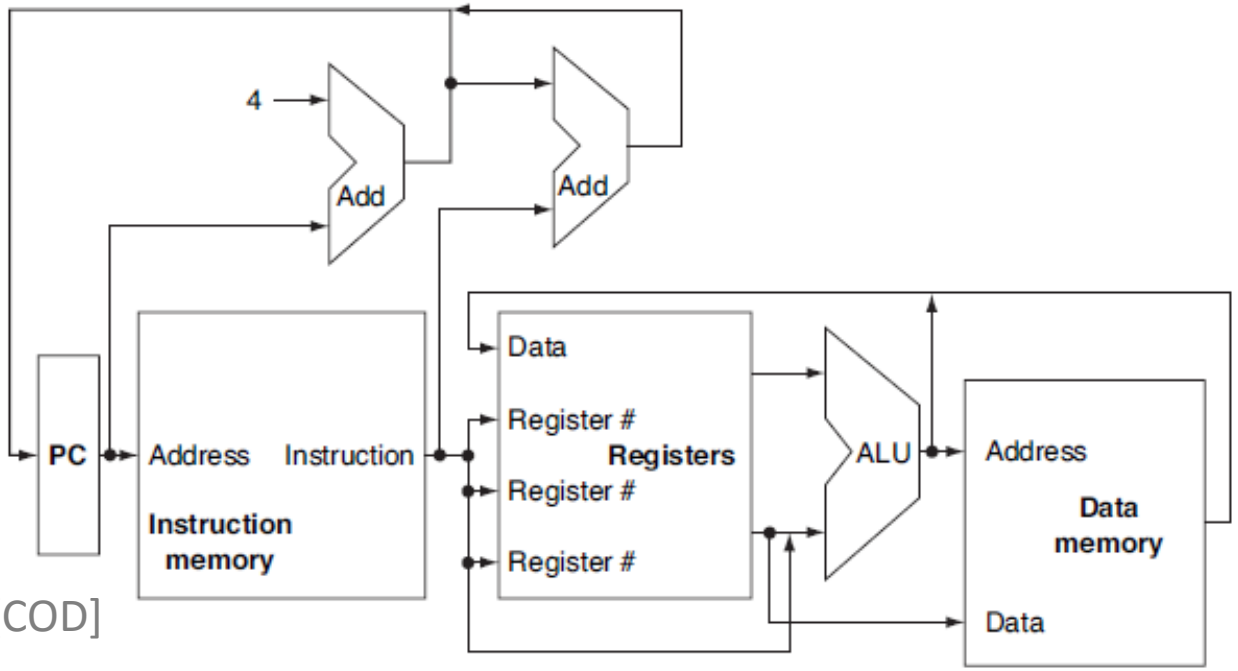
[Pentru mai multe informaţii despre File Register şi ALU vezi prelegerile anterioare]



[COD]

Instrucţiuni load / store

- Pentru **instrucţiunile load şi store** (în format **I**) sunt necesare:
  - ✓ **Memoria de date:** locul unde se păstrează datele şi de unde sunt preluate valorile pentru încărcare în regiştrii sau unde sunt stocate valorile preluate din regiştrii
  - ✓ **Sign Extend:** o componentă care realizează extinderea cu semn de la 16 la 32 de biţi (momentan nu apare pe schemă)



[COD]

Instrucţiuni load / store

- *Întrebare:* Unde ar trebui sa apară componenta Sign Extended?

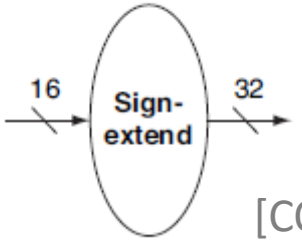
Hint:

`lw rt, address`

0x23	rs	rt	Offset
6	5	5	16

`sw rt, address`

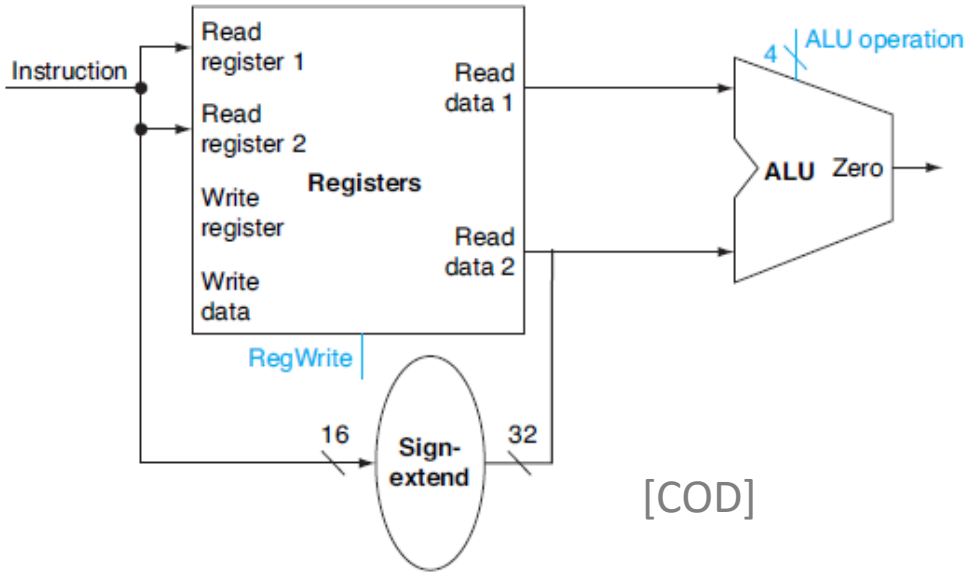
0x2b	rs	rt	Offset
6	5	5	16



[COD]

- *Răspuns:* Pentru a calcula adresa offset(rs), ALU realizează suma rs + offset

```
lw $t1, 4($t0)
```

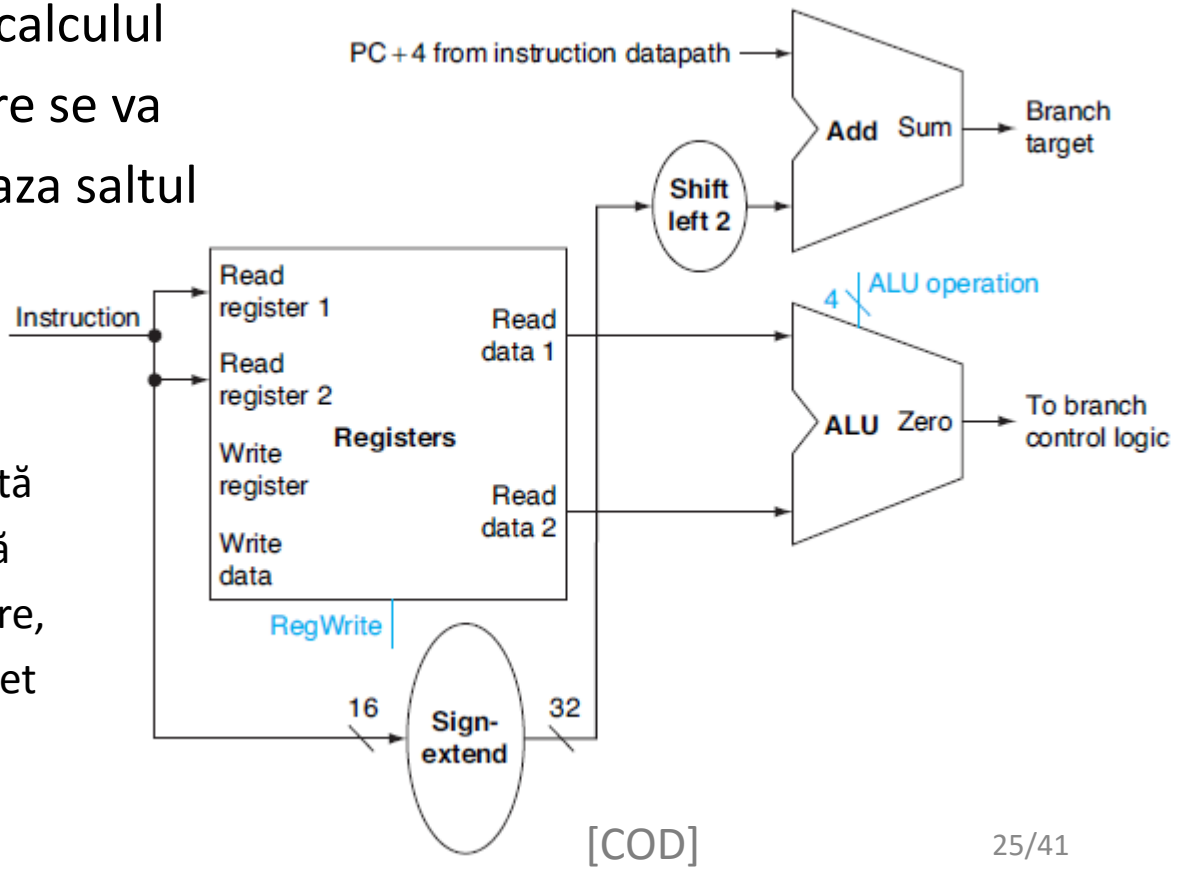


[COD]

Instrucţiuni condiţionate (beq)

- Pentru **instrucţiunile condiţionate beq** (în format **R**) sunt necesare:
  - ✓ **ALU:** folosit la calculul condiţiei de salt (realizeaza scăderea si verifică dacă rezultatul este 0, foloseste deci flagul zero al ALU)
  - ✓ **ALU (Add):** folosit la calculul adresei instrucţiunii care se va efectua dacă se realizeaza saltul (conditia de egalitate este indeplinită)

Atenţie! Pe implementarea prezentată, offsetul reprezintă numarul de instrucţiuni după adresa instrucţiunii următoare, i.e. PC devine (PC+4) + 4 offset



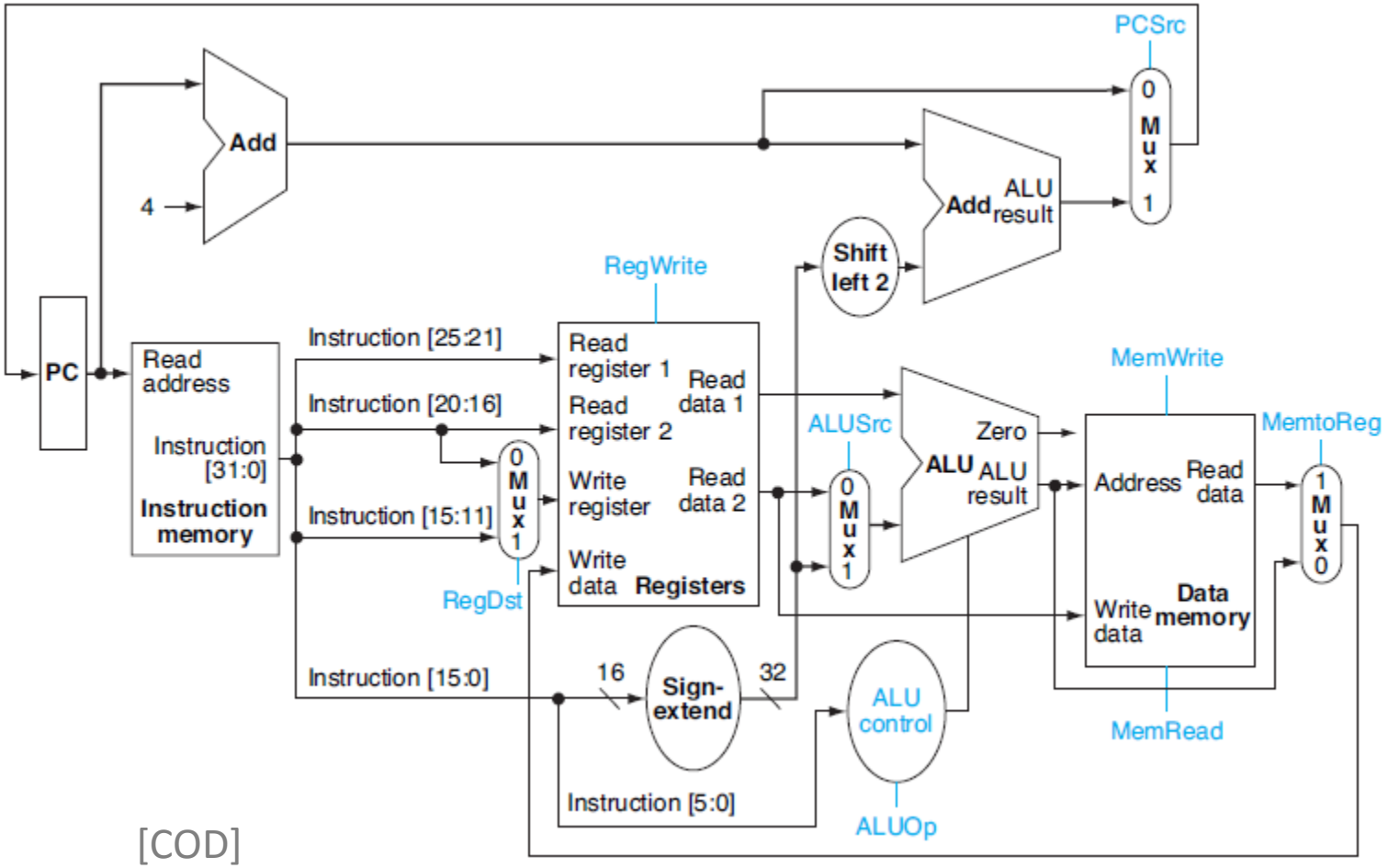
Implementare (I)

Pornim de la schema simplă şi introducem câteva elemente suplimentare:

- **MUX:** multiplexoare pentru selectarea unei intrări acolo unde sunt mai multe posibile; exemple:
  - ✓ operandul al doilea din ALU poate fi valoarea dintr-un registru pentru add sau offsetul pentru `lw` şi `sw`;
  - ✓ valoarea PC poate să fie PC + 4 pentru instrucţiuni necondiţionate sau când condiţia de salt nu este îndeplinită sau (PC+4) + 4 offset când condiţia de salt este îndeplinită)
- **semnale de control**: detaliate în slide-urile următoare
- **biţii exacţi ai instrucţiunii** care sunt folosiţi în diferite etape (pe diferite circuite); exemple:
  - ✓ **Instruction [15:0]** sunt ultimii 16 biţi din formatul instrucţiunii, adica biţii de adresă pentru `lw/sw` şi `beq`;
  - ✓ **Instruction [25:21]** sunt biţii pe care se reprezintă în binar primul registru sursă pentru `add`

Implementare (I)

- O primă implementare, fără instrucţiunea `j` şi unitate centralizată de control:



[COD]

Unitatea de control

- *PCSrc* : selectează sursa PC (0 dacă nu se face salt; 1 dacă se face salt)
- *RegWrite* : indică dacă se permite scrierea în regiştrii (1 pentru scriere, 0 altfel)
- *MemWrite* : indică dacă se permite scrierea în memorie (1 pentru scriere, 0 altfel)
- *MemRead* : indică dacă se permite citirea din memorie (1 pentru citire, 0 altfel)
- *ALUSrc* : indică sursa celui de-al doilea operand în ALU
- *RegDst* : indică sursa registrului destinaţie în ALU
- *MemtoReg* : indică sursa valorii scrise în registrul destinaţie (0 pentru rezultatul ALU, 1 pentru valoarea citită din memorie)
- *ALUOp* : indică operaţia efectuată de ALU, detaliată ulterior

Unitatea de control

- Am stabilit într-o prelegere anterioară unitatea de control a ALU:

ALU operation	Operaţie
0000	and
0001	or
0010	add
0110	substract
1100	nor
0111	set on less than

ALUOp	Operaţie	Utilizare
00	add	lw, sw
01	substract	beq
10	func Instruction[5:0]	Instr.logice si aritmetice

- Introducem acum doar 2 biţi de control (*ALUOp*) care indică direct operaţia (00, 01) sau aceasta depinde de operaţia codată în instrucţiune (10)

Unitatea de control

- Mai exact, componenta *ALU Control* satisface:

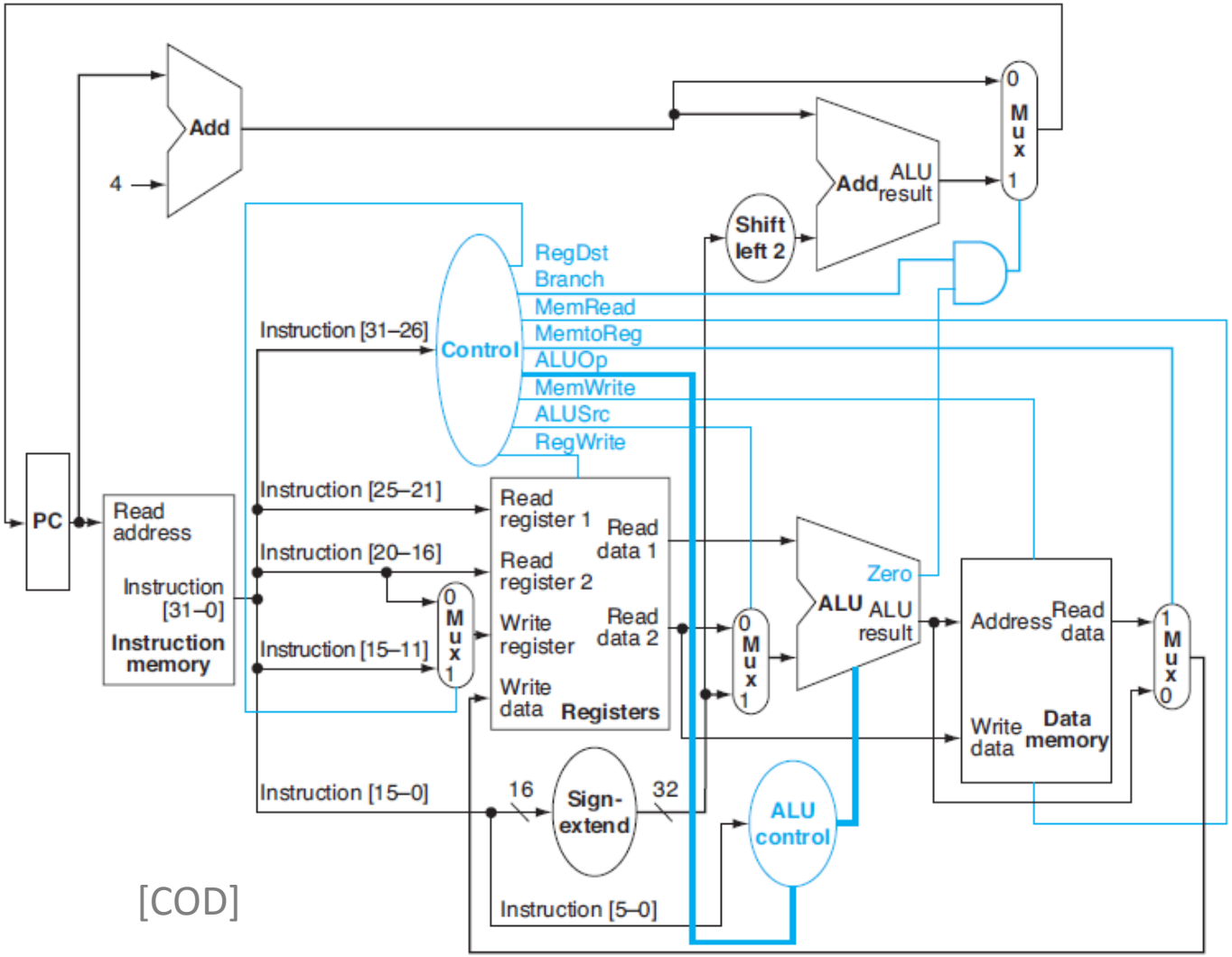
ALUOp	Operaţie	ALU operation	Funct	Instructiune
00	add	0010	xxxxxx	lw
00	add	0010	xxxxxx	sw
01	substract	0110	xxxxxx	beq
10	and (func)	0000	100100	and
10	or (func)	0001	100101	or
10	add (func)	0010	100000	add
10	substract (func)	0110	100010	substract
10	slt (func)	1100	101010	slt

[xxxxxx apare pentru instructiunile care nu sunt in format R, deci nu au func]

- *ALU Control* se poate implementa ca un circuit 0-DS

Unitatea de control

- Introducem o unitate de control centralizat *Control*:



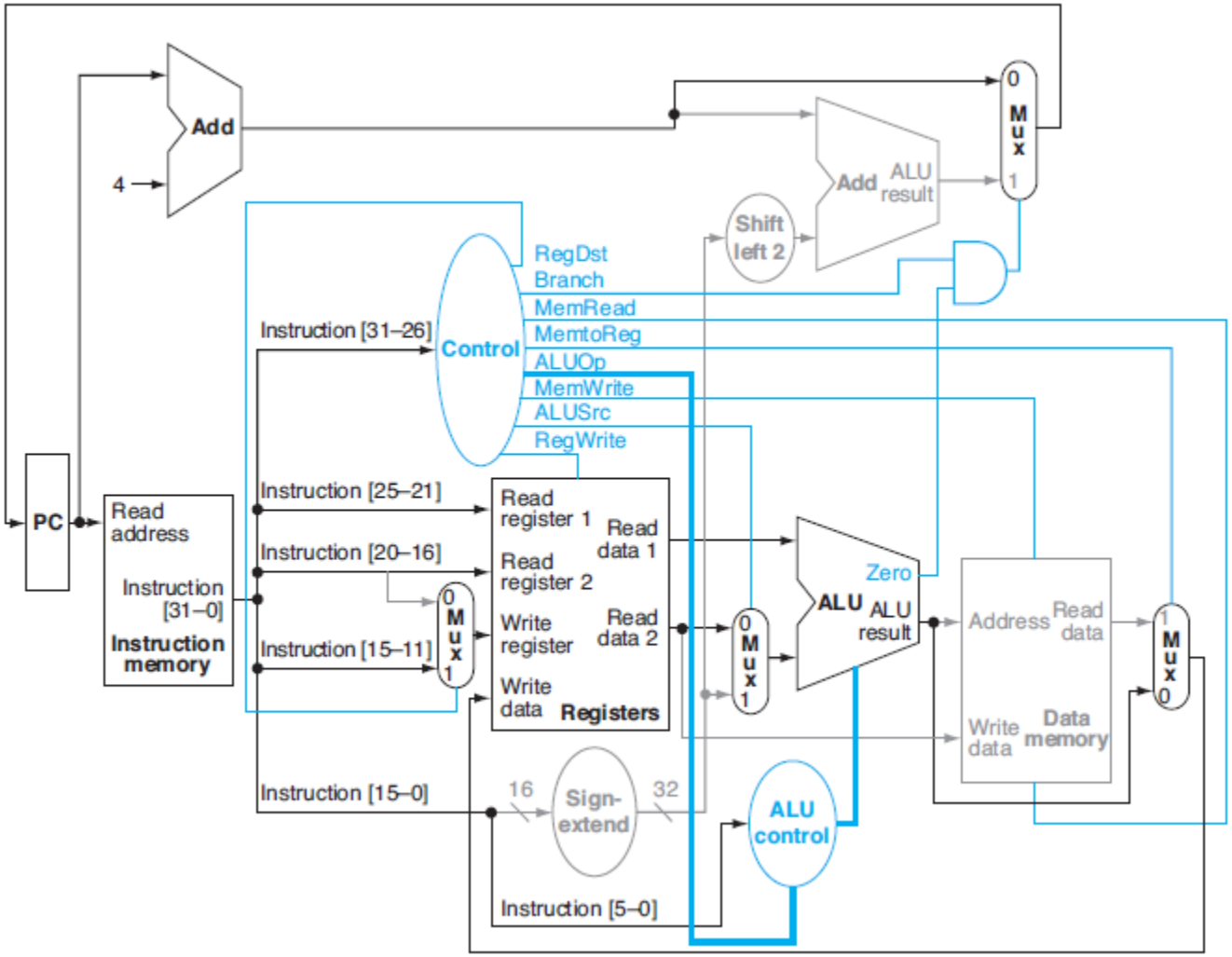
Unitatea de control

- Unitatea centralizată de control (*Control*) primeşte ca intrare Instruction[31:26], deci biţii op din instrucţiune şi scoate la ieşire semnalele necesare pentru fiecare tip de instrucţiune:

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

[COD]

Procesarea intrucţiunii add



Procesarea intrucţiunii add

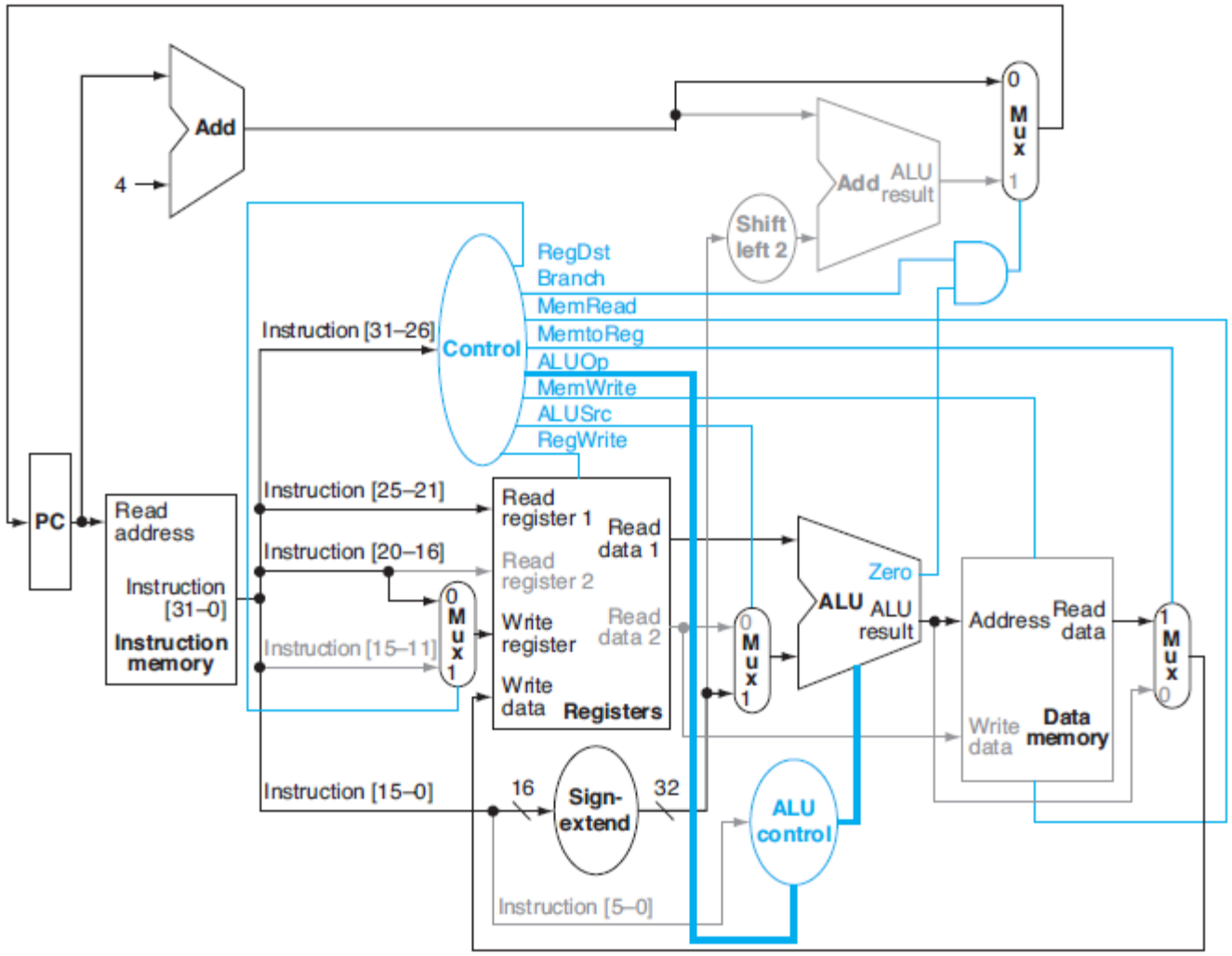
1. Se încarcă intrucţiunea şi se incrementează PC (cu 4)

2. Regiştrii operanzi sunt citiţi din Register File. Unitatea de control calculează semnalele de control

3. ALU foloseşte func (Instruction[5:0]) şi calculează ALU Result

4. ALU Result este scris în fişierul destinaţie (Instruction[15:11])

Procesarea intrucţiunii lw



Procesarea intrucţiunii lw

1. Se încarcă intrucţiunea şi se incrementează PC (cu 4)

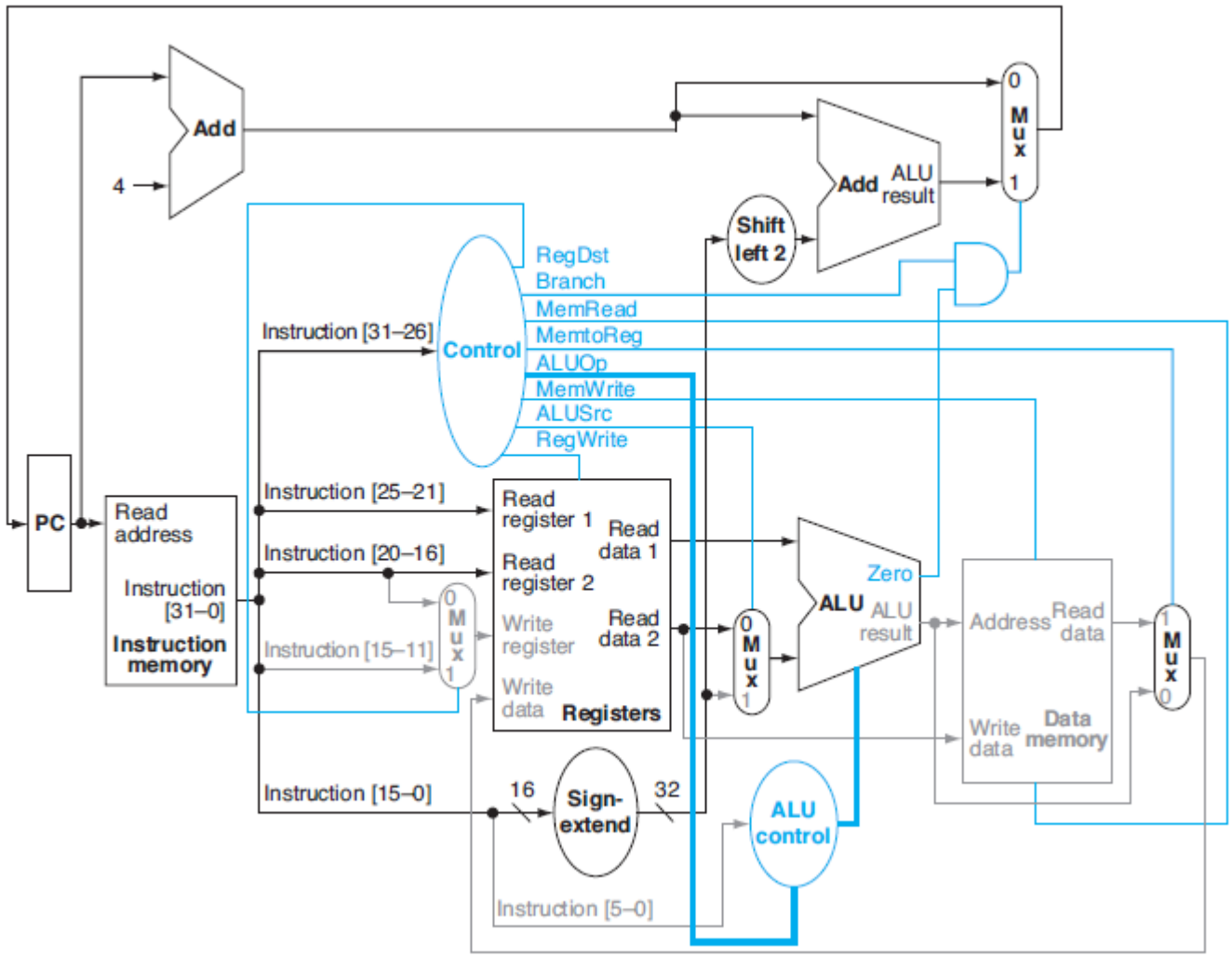
2. Un registru (Intruction[25:21]) este citit din Register File

3. ALU calculează suma dintre valoarea registrului citit şi offset (Instruction[15:0]), extins de la 16 la 32 de biţi

4. ALU Result este adresa din memoria de date

5. Valoarea citită de la adresa indicată este scrisă în fişierul destinaţie (Instruction[20:16])

Procesarea intrucţiunii beq



Procesarea intrucţiunii beq

1. Se încarcă intrucţiunea şi se incrementează PC (cu 4)

2. Regiştrii operanzi sunt citiţi din Register File

3. ALU scade valorile din cei 2 regiştrii. PC+4 este adunată la offset (Instruction[15:0]), extins de la 16 la 32 de biţi şi shiftat la stânga cu 2 (înmulţit cu 4) pentru a determina adresa de salt

4. Flag-ul Zero al ALU indică ce adresă se stochează în PC

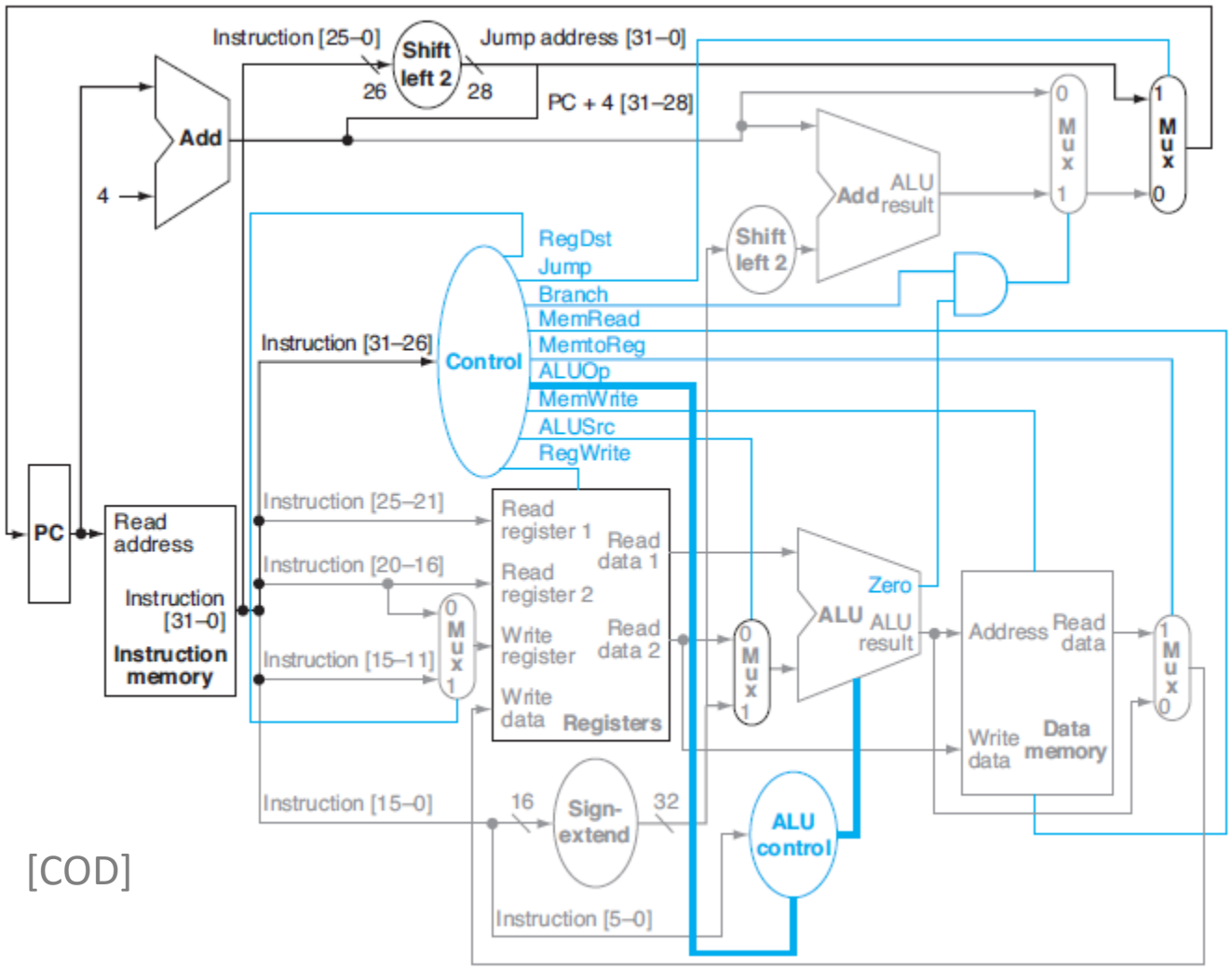
Implementare (I)

- Instrucţiunea necondiţionată jump (`j`) se introduce imediat prin adaugarea adresei:
  - ✓ adresa este exprimată în word-uri, deci se shiftează la stânga cu 2 (i.e. se înmulţeşte cu 4)
  - ✓ primele poziţii ale adresei se copiază din PC + 4

- Apare semnalul de control *Jump* (activ doar pentru instrucţiunile de tip J) şi un multiplexor pentru selecţia adresei stocate in PC

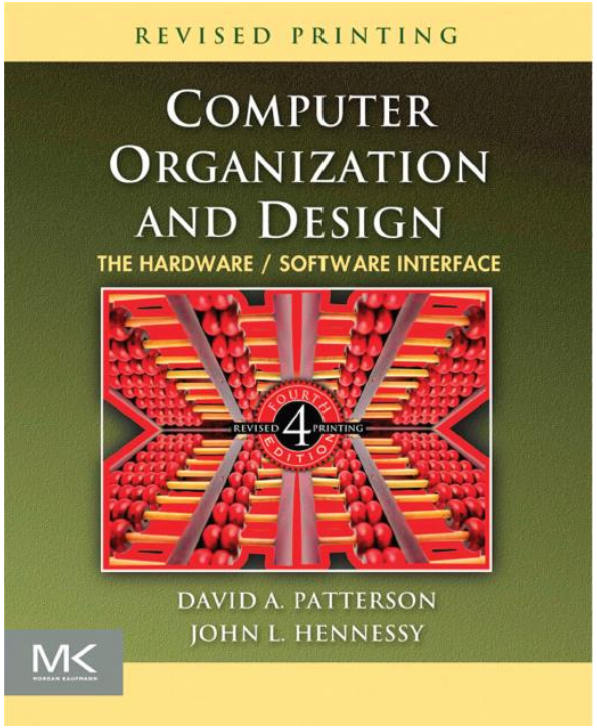
Procesarea intrucţiunii j

- O primă implementare finală, cu instrucţiunea `j` şi unitate centralizată de control:



[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/