

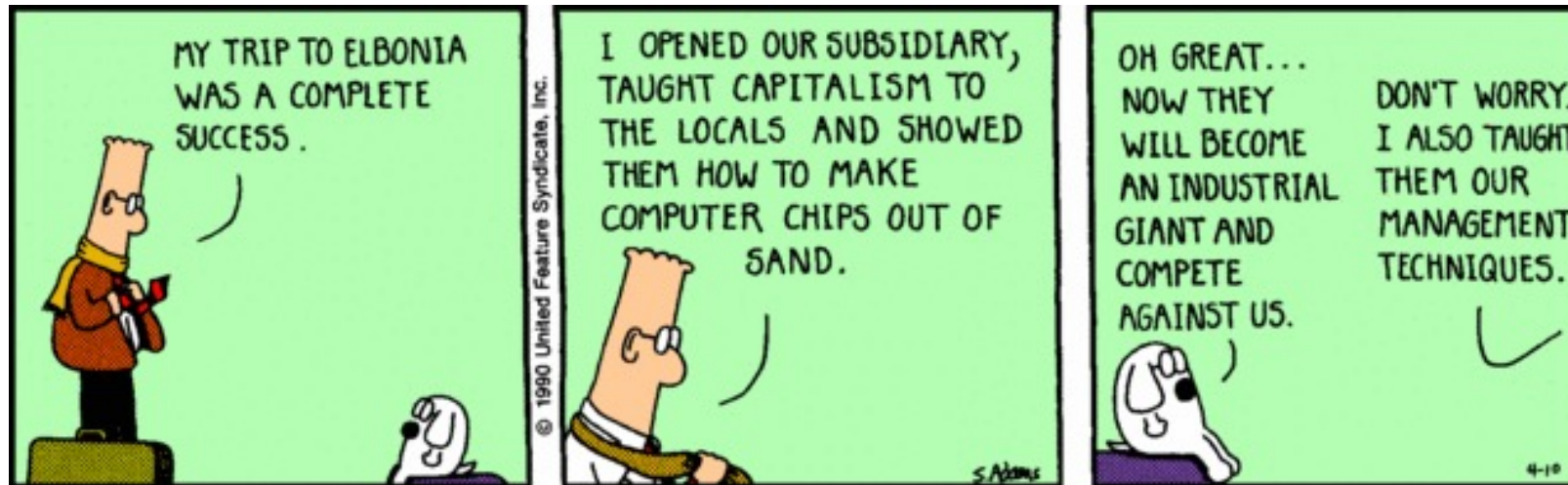
Calculatoare Numerice

– Cursul 7 –

Implementarea unei mașini de calcul

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day



<http://dilbert.com/strips/comic/1990-04-10/>

Din cursurile trecute

- Computer Architecture >> ISAs & RTL
 - Cursul se ocupă de interacțiunea hardware/software și de proiectarea nivelurilor potrivite de abstractizare
- Tehnologia și aplicațiile ei modelează arhitectura sistemelor de calcul
 - Trecutul poate da lecții pentru dezvoltarea tehnicii
- Primii 130 de ani de CompArch, de la Babbage la IBM 360
 - Evoluția de la calculatoare mecanice la mașini complet programabile
 - Schimbarea rapidă în timpul WWII (anii 45), tranziție de la procesoare electro-mecanice la electronice
- Costul software development devine o problemă pentru dezvoltarea arhitecturilor de calcul (era nevoie de compatibilitate)
- IBM 360 introduce noțiunea de “familie de calculatoare” cu același ISA care rulează pe mașini complet diferite hardware
 - Șase mașini diferite lansate în aceeași zi (7 aprilie 1964)
 - “Future-proofing” pentru generațiile viitoare

IBM 360: Implementarea inițială

	<i>Model 30</i>	<i>...</i>	<i>Model 70</i>
<i>Memory</i>	8K - 64 KB		256K - 512 KB
<i>Datapath</i>	8-bit		64-bit
<i>Circuit Delay</i>	30 nsec/level		5 nsec/level
<i>Local Store</i>	Main Store		Transistor Registers
<i>Control Store</i>	Read only 1~sec		Conventional circuits

Arhitectura setului de instrucțiuni (ISA) IBM 360 ascundea complet diferențele de tehnologie dintre diferitele modele.

Milestone: Primul ISA proiectat ca o interfață portabilă între hardware și software!

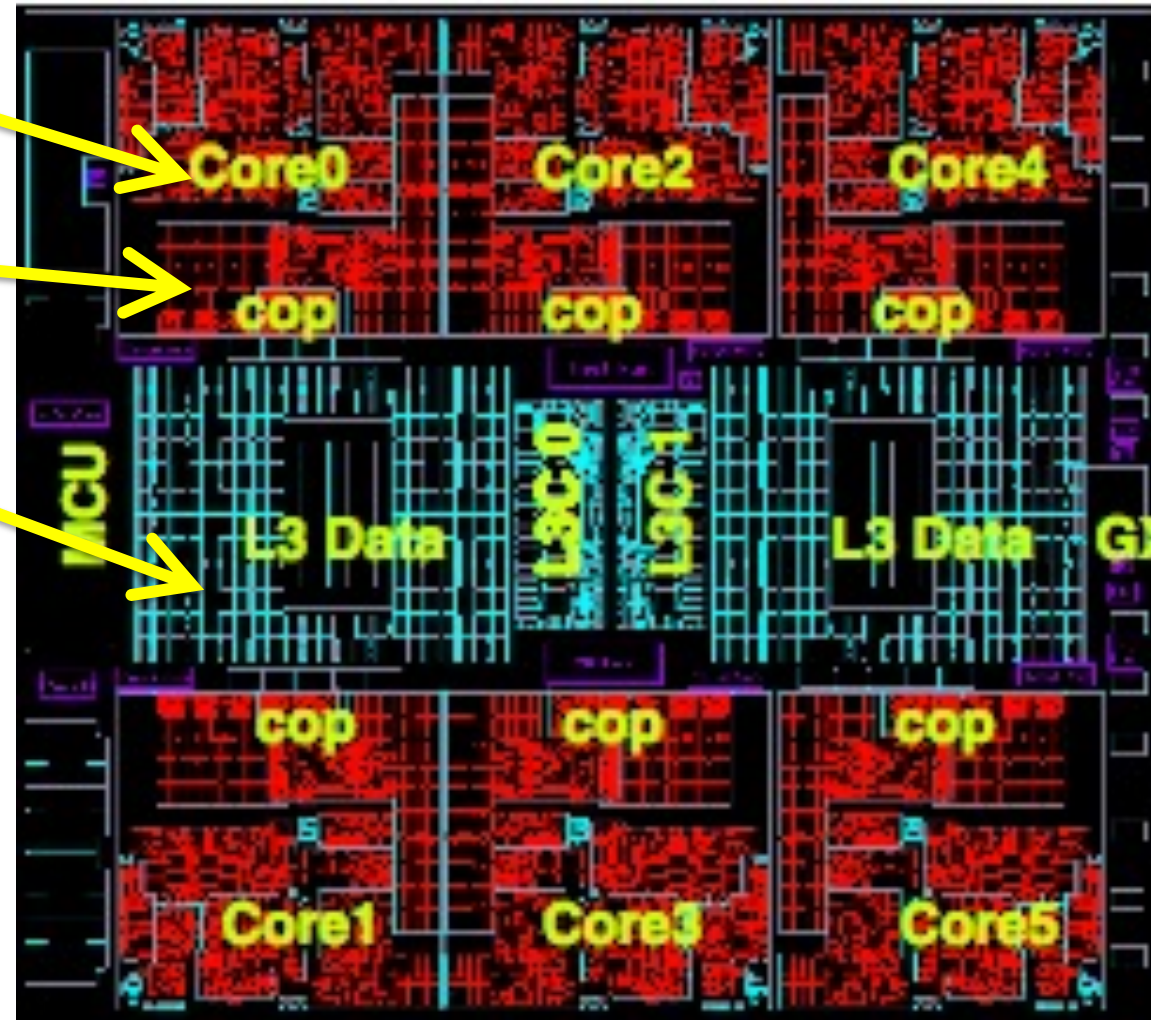
IBM 360 Supraviețuiește și azi: z12 Mainframe Processor

6 Cores @ 5.5 GHz

Coprosesor special-
purpose pe fiecare
core

48MB Level-3
cache on chip

32nm SOI Technology
2.75 miliarde tranzistoare
23.7mm x 25.2mm
15 straturi de metal
12.3 km de conexiuni!
10,000 pini de alimentare (!)
1,071 I/O pins



[IBM HotChips24 presentation, August 28, 2012]

Instruction Set Architecture (ISA)

- Punctul de contact dintre software și hardware
- Descrisă în mod tipic prin punerea la dispoziția programatorilor a registrelor, memoriei și a semanticii instrucțiunilor care operează pe mașina respectivă
- IBM 360 a fost prima familie de mașini de calcul care a separat ISA de implementarea hardware propriu-zisă (aka. *microarhitectură*)
- Mai multe implementări posibile pentru aceeași ISA
 - E.g., sovieticii au construit clone compatibile ale IBM360, precum și Amdahl, după plecarea din IBM.
 - E.g.2., astăzi, procesoarele Intel și AMD folosesc ISA x86-64.
 - E.g.3: majoritatea telefoanelor mobile folosesc ISA ARM cu implementări hardware de la mai multe companii, printre care și TI, Qualcomm, Samsung, Marvell, etc.

Maparea ISA pe microarhitectură

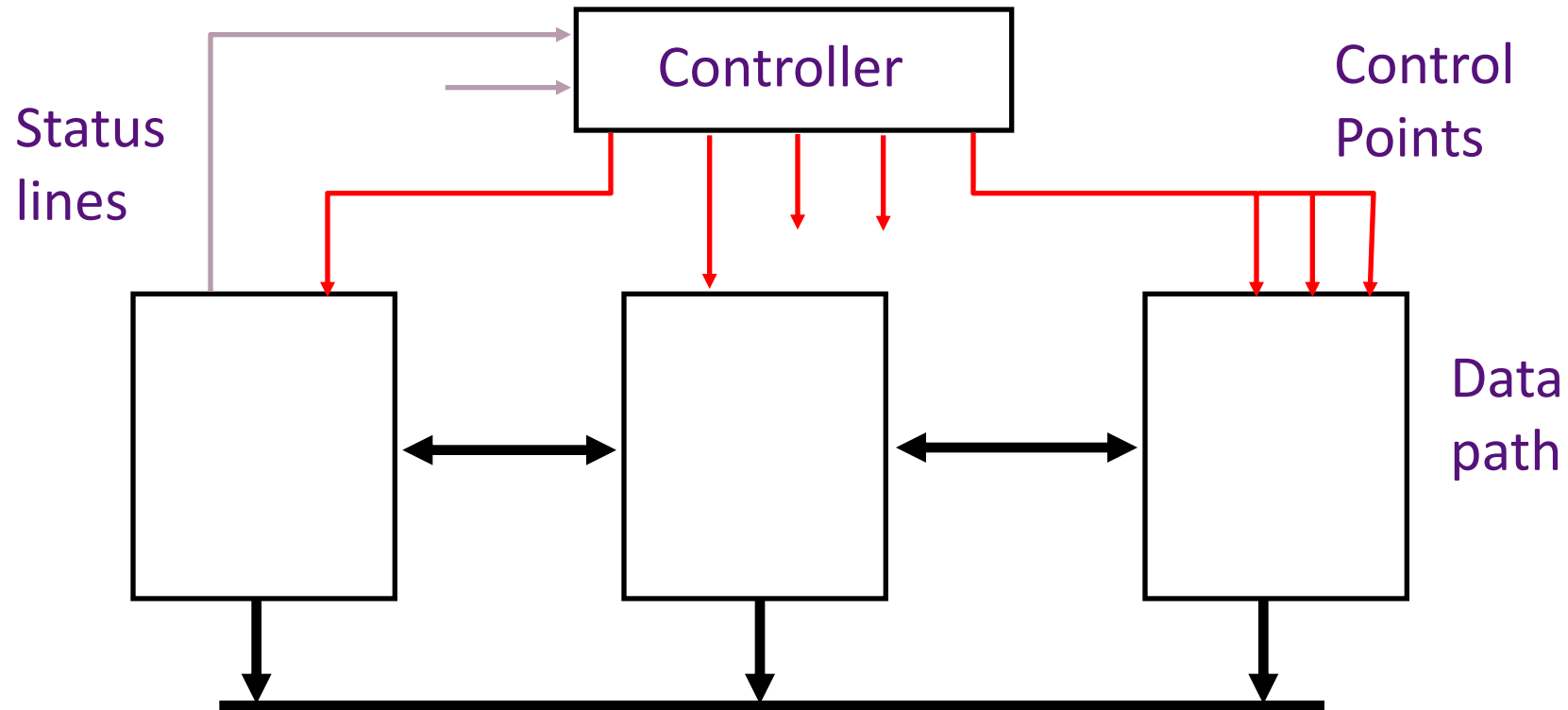
- ISA poate fi proiectată ținând cont tipul de microarhitectură:
 - Acumulator ⇒ hardwired, unpipelined
 - CISC ⇒ microcoded
 - RISC ⇒ hardwired, pipelined
 - VLIW ⇒ fixed-latency in-order parallel pipelines
 - JVM ⇒ software interpretation
- Dar poate fi implementată folosind orice fel de microarhitectură
 - Intel Ivy Bridge: hardwired pipelined CISC (x86)
 - Simics: Software-interpreted SPARC RISC machine
 - ARM Jazelle: Procesor JVM hardware
 - În acest curs: mașină RISC-V cu microcod

Microprogramarea în cursul de azi

- Pentru a arăta cum pot fi construite procesoare mici care rulează un ISA complex
- Pentru a vă ajuta să înțelegeți de ce există mașini CISC*
- Pentru că este încă folosită la scară largă (IBM360, x86, PowerPC)
- Servește ca o introducere (mai) blândă în structura mașinilor de calcul
- Pentru a vă face să înțelegeți de ce tehnologia s-a centrat pe RISC*

* “CISC”/”RISC” sunt denumiri mult mai noi decât arhitecturile la care fac referință.

Microarchitctură: *Implementarea unei ISA*



Structură: Cum sunt interconectate componentele.

Statică

Comportament: Cum se mișcă datele între diferitele componente

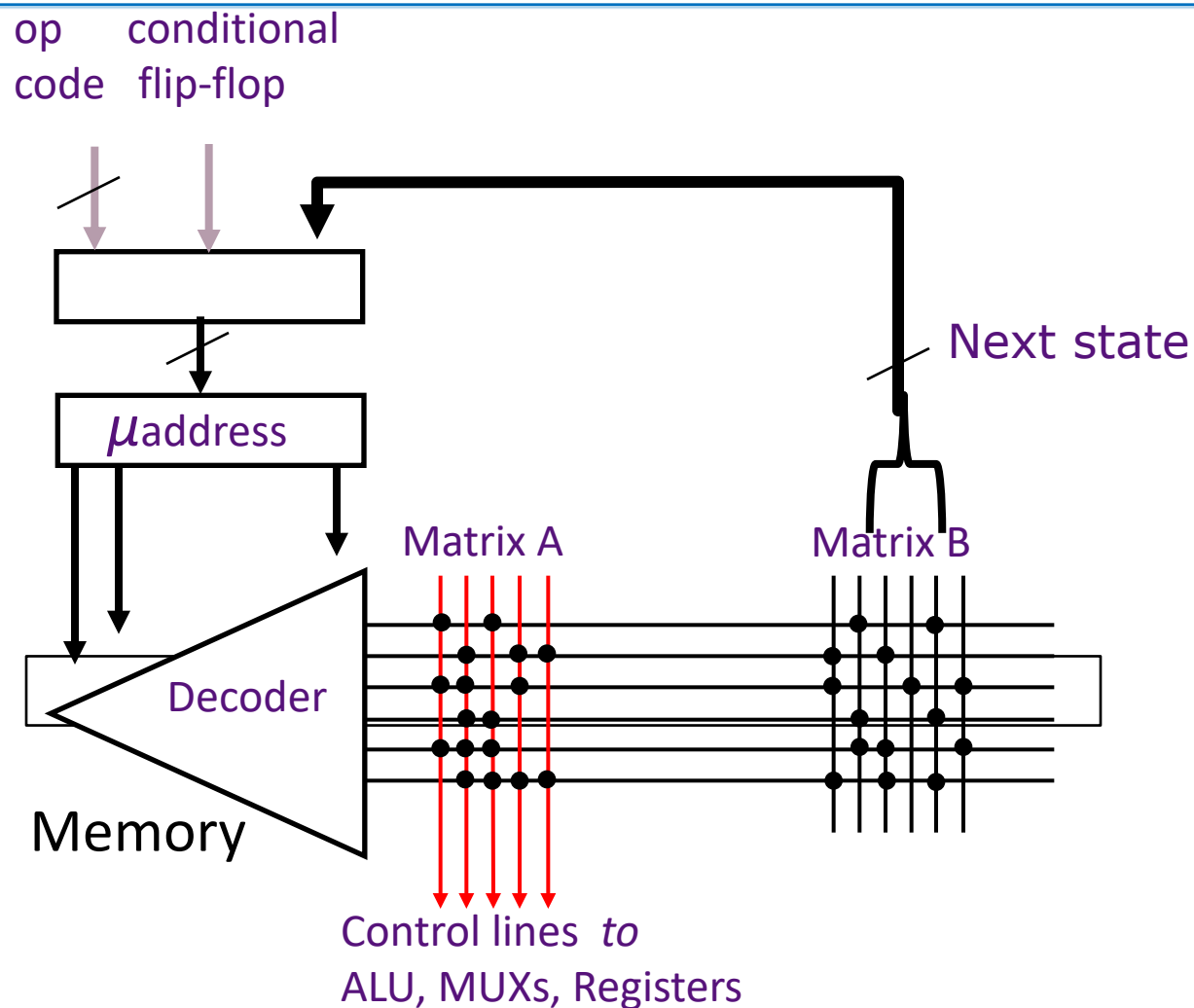
Dinamică

Unitate control microprogramat

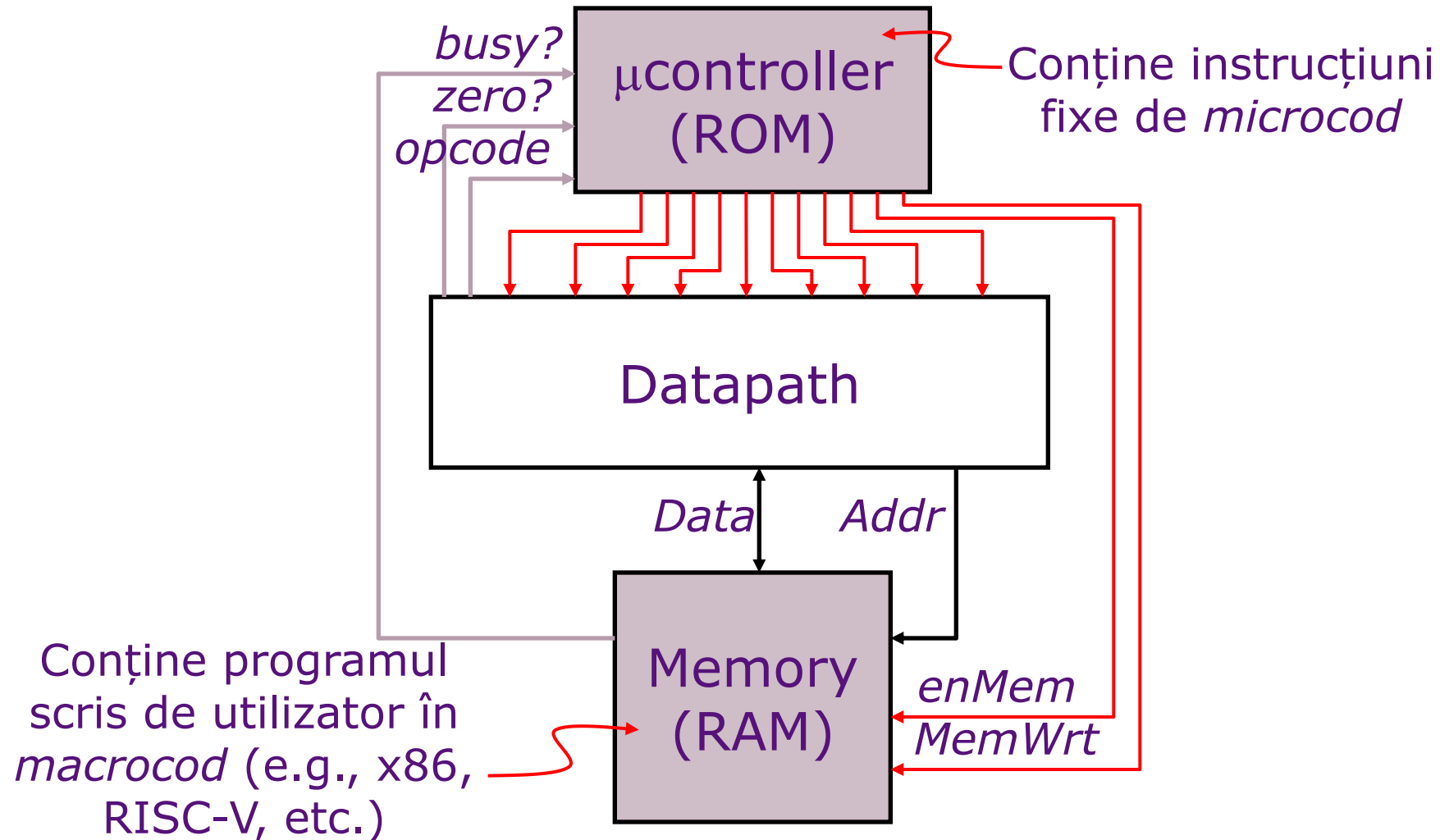
Maurice Wilkes, 1954

Folosită prima dată în
EDSAC-2, 1958

Încorporează
tabela de stări
pentru logica de
control într-o
memorie



Microarchitctură microcodată



RISC-V ISA

- Design RISC de la UC Berkeley
- ISA complet dar de mici dimensiuni și deschis
- Nu este supra-optimizat pentru un anumit stil de implementare
- Două variante, pentru 32 și 64 de biți
 - RV32 & RV64
- Proiectat pentru multiprocesoare
- Codificare eficientă a instrucțiunilor
- Ușor de extins/restrâns pentru educație/cercetare

- Pentru acest curs vom folosi o arhitectură RISC-V pe 32 de biți, foarte similară cu a procesorului MIPS

RV32 Processor State

Program counter (**pc**)

32x32-bit registre pt întregi (**x0-x31**)

- **x0** conține întotdeauna 0

32 registre floating-point (FP) (**f0-f31**)

- fiecare poate conține o valoare single- sau double-precision FP (32-bit sau 64-bit IEEE FP)

FP status register (**fsr**), folosit pentru FP rounding mode & raportare excepții

XPRLN-1	0
x0 / zero	
x1 / ra	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	
x10	
x11	
x12	
x13	
x14	
x15	
x16	
x17	
x18	
x19	
x20	
x21	
x22	
x23	
x24	
x25	
x26	
x27	
x28	
x29	
x30	
x31	

XPRLN-1	0
pc	XPRLN

63	0
f0	
f1	
f2	
f3	
f4	
f5	
f6	
f7	
f8	
f9	
f10	
f11	
f12	
f13	
f14	
f15	
f16	
f17	
f18	
f19	
f20	
f21	
f22	
f23	
f24	
f25	
f26	
f27	
f28	
f29	
f30	
f31	

31	0
fsr	32

Formatul instrucțiunilor

High-level language statement:

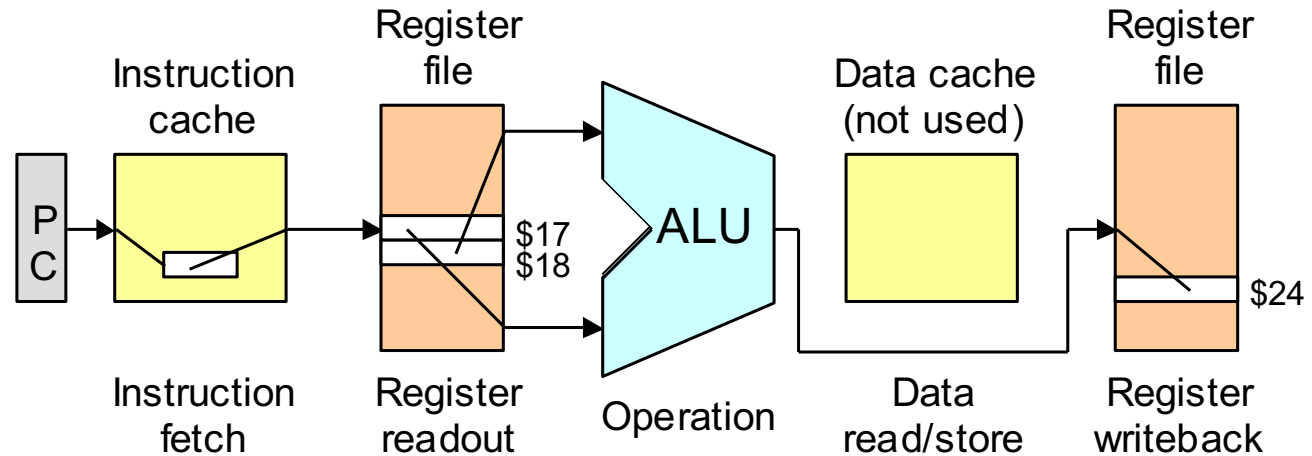
`a = b + c`

Assembly language instruction:

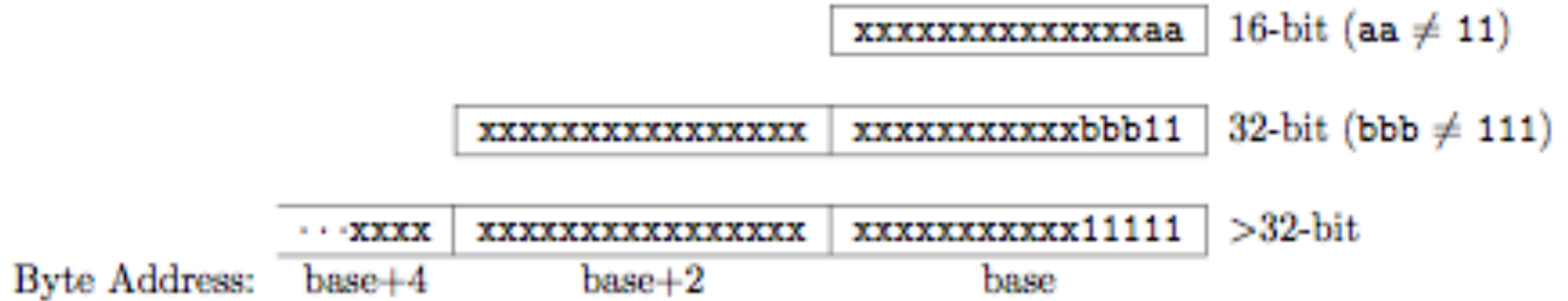
`add $t8, $s2, $s1`

Machine language instruction:

000000 10010 10001 11000 00000 100000
 ALU-type Register Register Register Unused Addition
 instruction 18 17 24 opcode

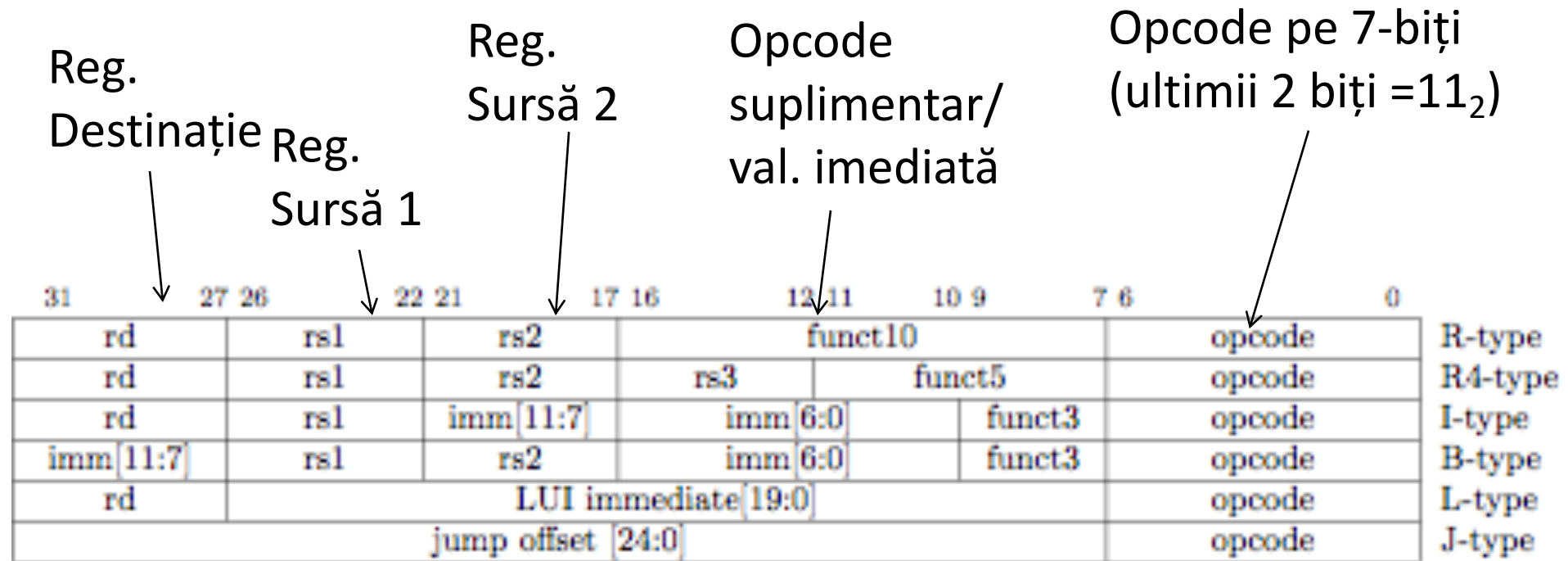


RISC-V Codificarea Instrucțiunilor

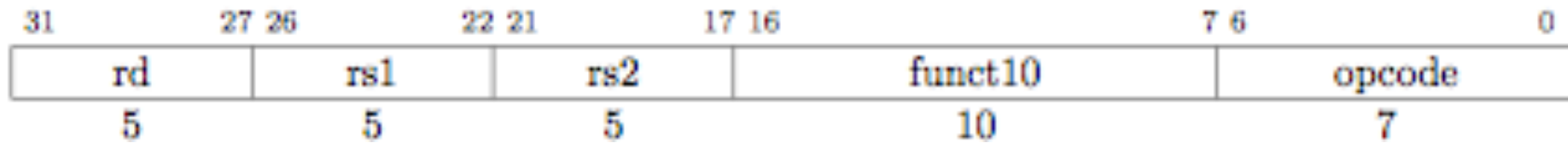


- Suport pentru instrucțiuni de lungime variabilă.
- Setul de bază (RV32) are întotdeauna instrucțiuni de dimensiune fixă, cu ultimii doi biți = 11_2
- Toate instrucțiunile de branch și jump au granularitate de 16 biți (chiar și în ISA de bază, unde toate instrucțiunile au lungimea de 32 de biți)

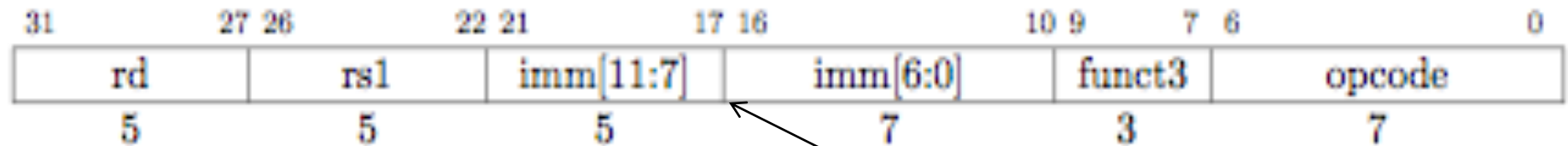
RISC-V Formatul Instrucțiunilor



R-Type/I-Type/R4-Type Formats



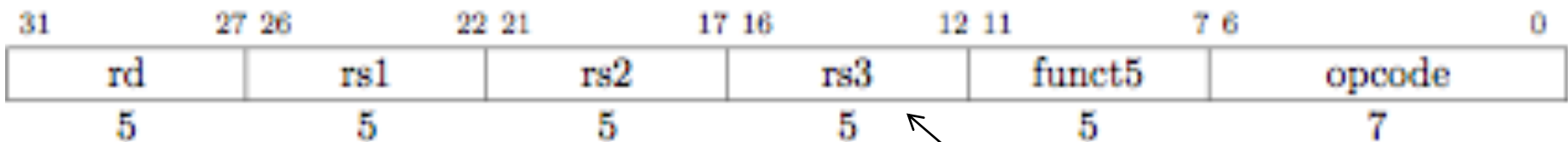
Operații Reg-Reg în ALU



Operații Reg-Imm ALU

Instrucțiuni Load, adresare (rs1 + imediat)

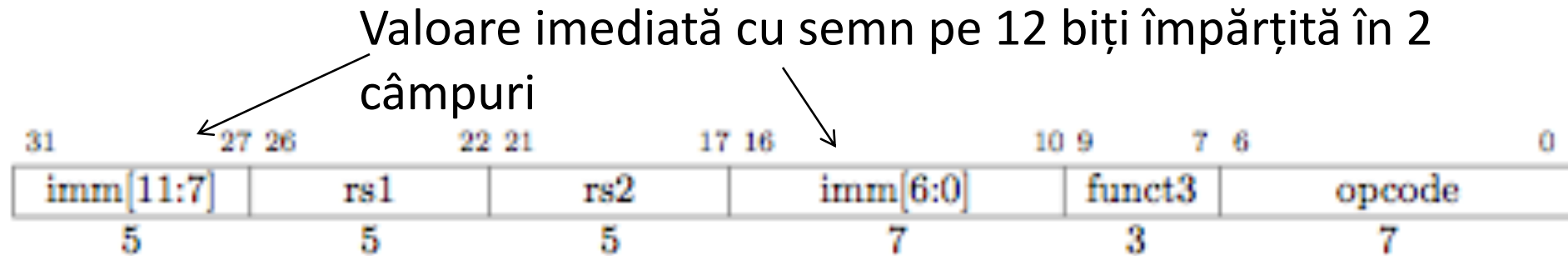
12-bit signed immediate



Folosite doar pentru adunări și înmulțiri în virgulă mobilă

Reg. Source 3

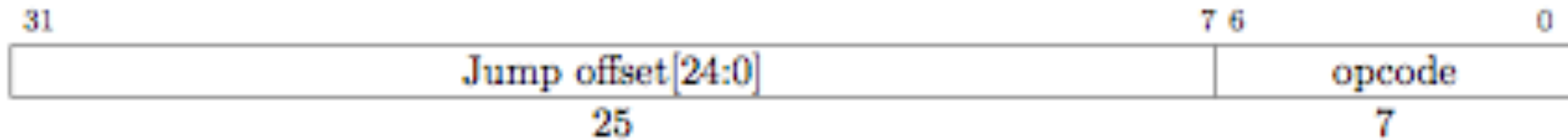
B-Type



Folosită pt. Branch, comparații de registre - $PC + (\text{immediat} \ll 1)$

Instrucțiuni Store, adresă ($rs1 + \text{immediat}$), date în rs2

J-Type



“J” Jump necondiționat, PC+offset target

“JAL” Jump and link, scrie PC+4 în **x1**

Formatul datelor și adresarea memoriei

Tipuri de date:

8-b Byte, 16-b Half word, 32-b word & 64-b double word

Remarci:

- *Adresarea octeților*

Little Endian

*Most Significant
Byte*

*Least Significant
Byte*



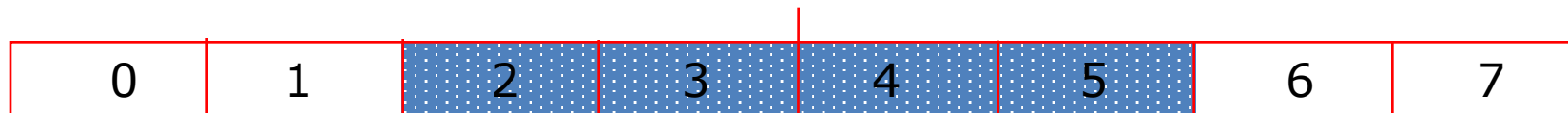
Big Endian



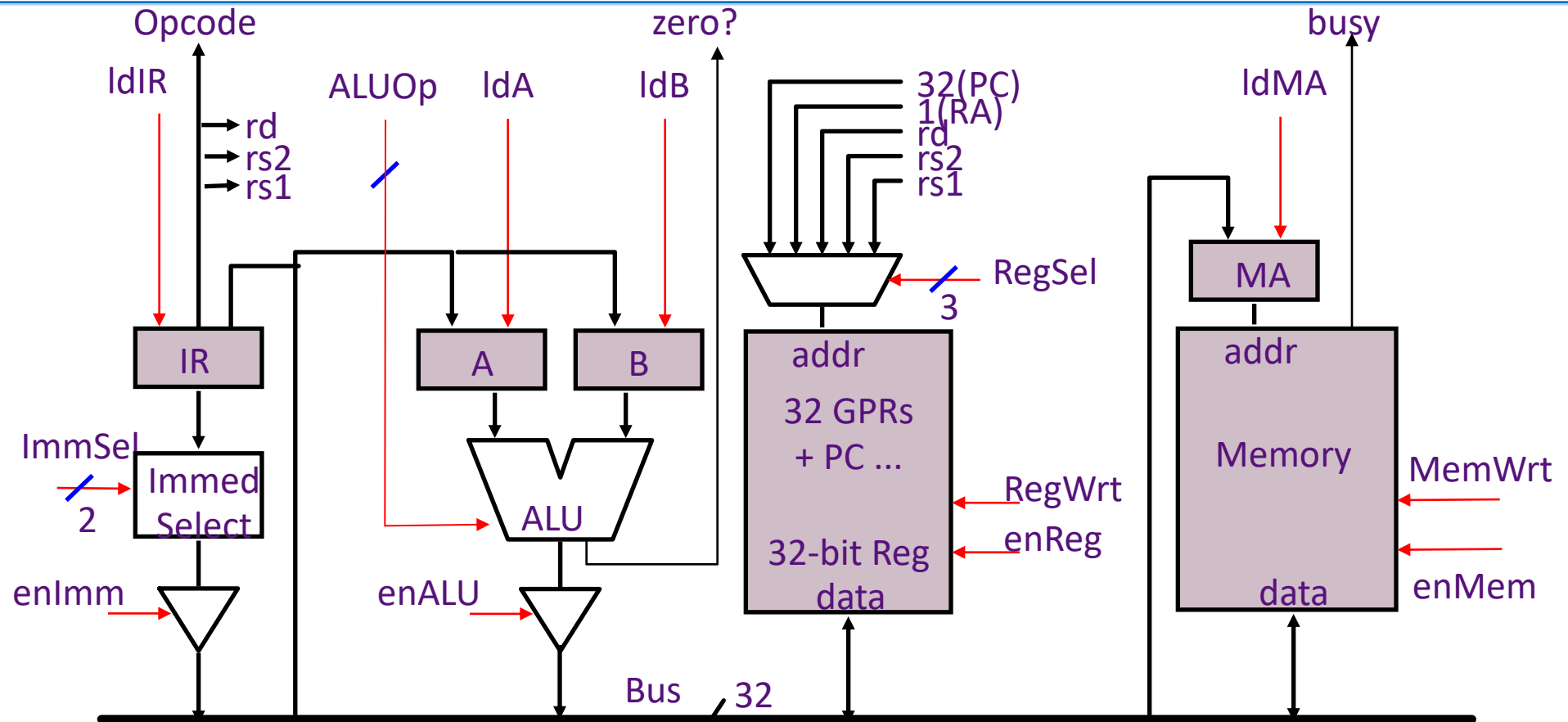
- *Alinierea cuvintelor*

Presupunem că memoria e organizată în cuvinte de 32 de biți.

Poate o adresă de cuvânt să înceapă numai la 0, 4, 8, ?



RISC-V – descriere generală



Microinstrucțiune: register to register transfer (17 semnale de control)

MA → PC se traduce prin RegSel = PC; enReg=yes; IdMA= yes
 B ← Reg[rs2] se traduce prin RegSel = rs2; enReg=yes; IdB = yes

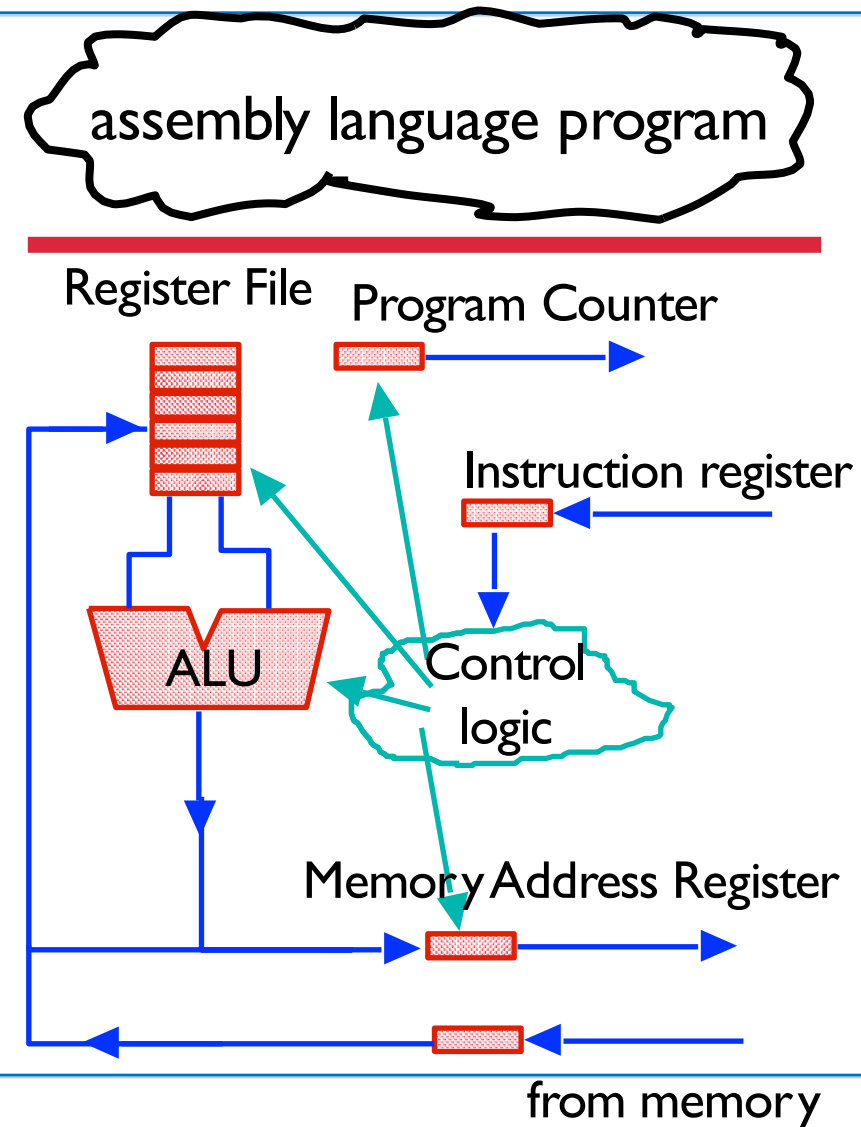
The Big Picture

Limbaj asamblare

Interfața pe care arhitectura o prezintă utilizatorului

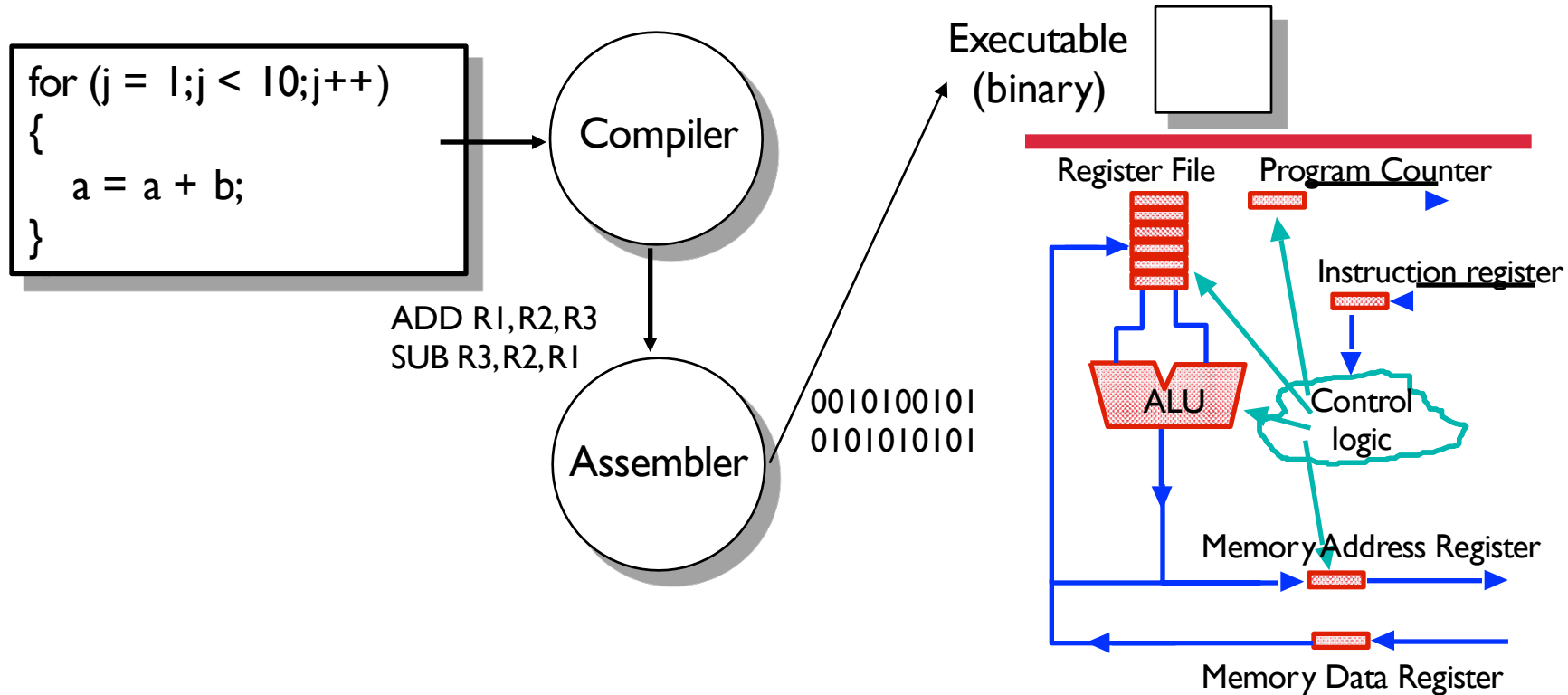
Instrucțiuni "Low-level" ce folosesc memoria și căile de date pentru a efectua operațiile de bază

- aritmetice: add, sub, mul, div
- logice: and, or, shift
- data transfer: load, store
- (un)conditional branch: jump, branch on condition



Niveluri de abstractizare

- De obicei, oamenii nu scriu software în cod mașină
- Folosesc limbaje de nivel înalt C, C++, JAVA, Python
- Limbajul de nivel înalt este translatat în assembly de un **compiler**
- Assembly este translatat în cod mașină de asamblor



Clase ISA

Memory to Memory Machines

Avem nevoie de stocare pentru variabile temporare

Memoria este lentă

Memoria este mare (mulți biți de adresă)

Registre

Registreele pot stoca **variabile temporare**

Registreele sunt **mai rapide** decât memoria

Traficul cu memoria este redus, iar execuția programului este accelerată (pentru că registreele sunt mia rapide ca memoria)

Densitatea codului este îmbunătățită (pentru că registreele sunt adresate mai ușor decât memoria)

Predomina GPR Machines

Toate mașinile de calcul folosesc registre generale

Avantajele registrelor

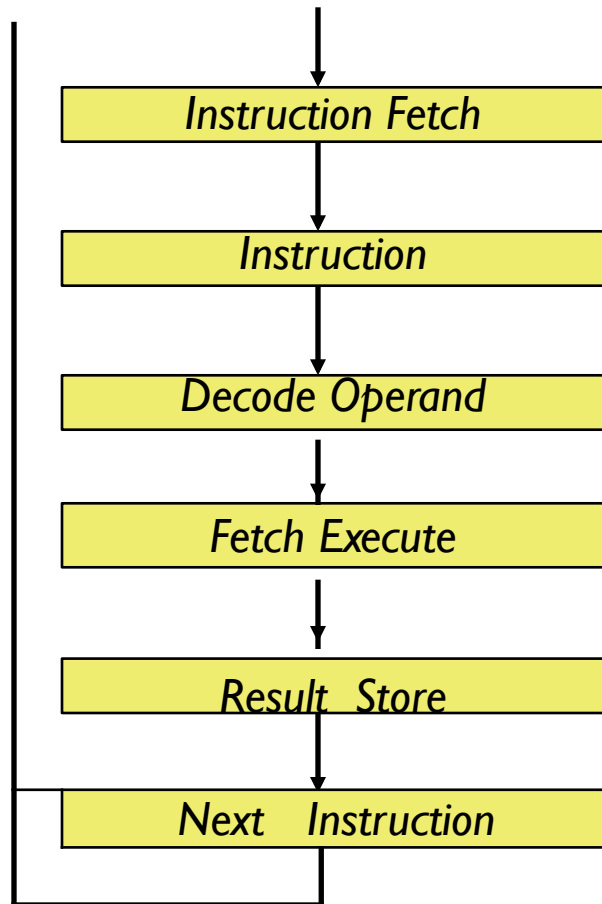
Registrele sunt **mai rapide** decât memoria

Traficul cu memoria este redus, iar execuția programului este accelerată (pentru că registrele sunt mai rapide ca memoria)

Registrele pot stoca **variabile**

Registrele sunt mai ușor de folosit pentru un compilator: e.g., $(A * B) - (C * D) - (E * F)$ poate executa în orice ordine vs. mașină cu stivă

Modelul execuției unei instrucțiuni



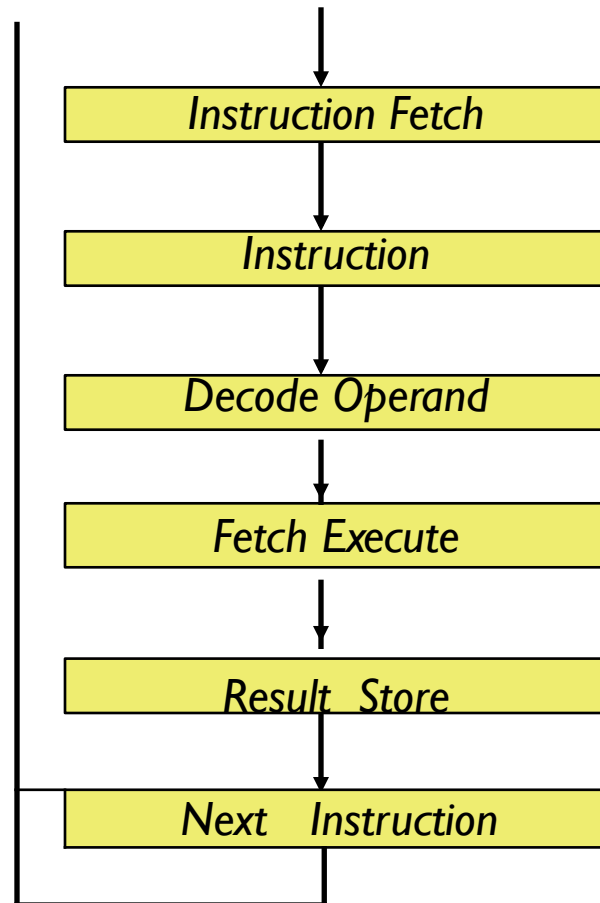
Model secvențial de execuție

Programul este o
secvență de execuție
Instrucțiunile sunt atomice
și executate secvențial

Conceptul programului stocat

Programul și datele sunt
stocate în memorie
Instrucțiunile sunt aduse din
memorie pentru execuție

Modelul execuției unei instrucțiuni



ISA Issues

Get instruction from memory

Instruction Format/Encoding

Addressing Modes

Op-codes and Data Types

Addressing Modes

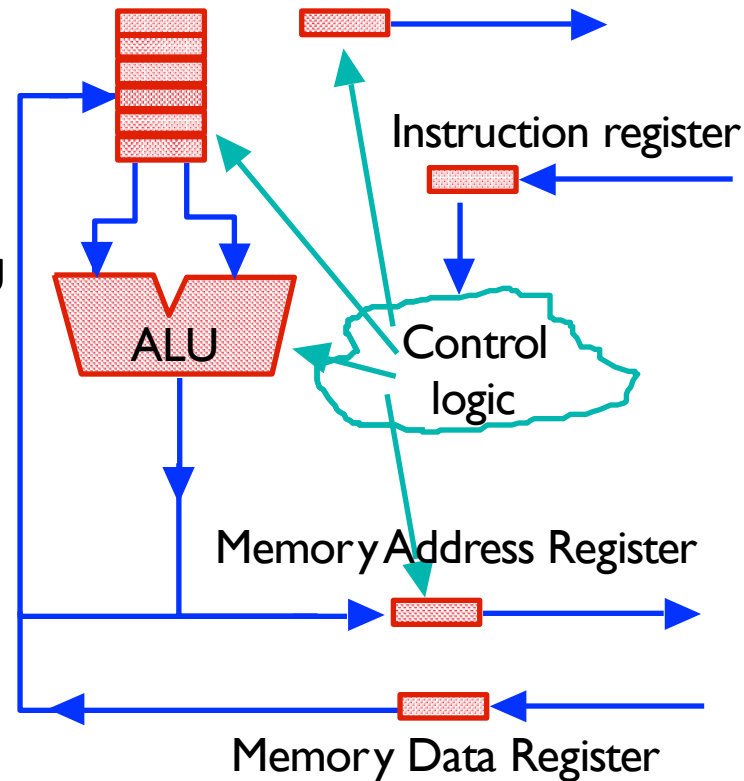
Instruction Sequencing

Execuția unei instrucțiuni

Secvențiatorul (FSM) face fetch la instrucțiune din memorie și o scrie în Instruction Register

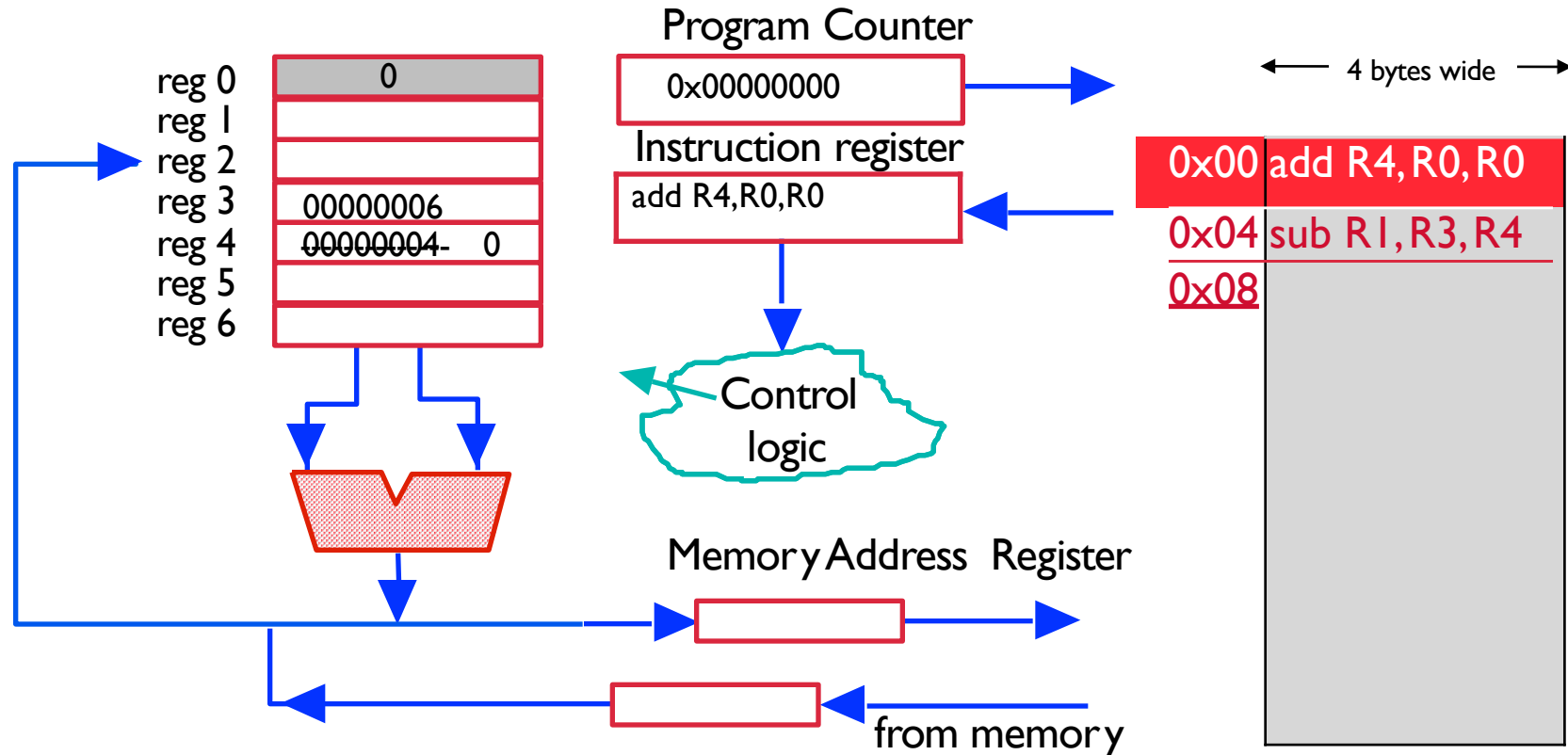
Logica de control **decodifică** instrucțiunea și emite comenzi către tabela de registre, ALU și celelalte registre

Dacă o operație ALU (e.g. add), datele merg din tabela de registre prin ALU și înapoi în tabela de registre



0x00	00101101
0x01	00100001
0x02	00110000
0x03	00001111
0x04	11010101
0x05	01010101
0x06	00101010
0x07	01010101
0x08	11110011
0x09	00111100
0x0A	00001100
0x0B	00000000
0x0C	00011000
0x0D	11111111

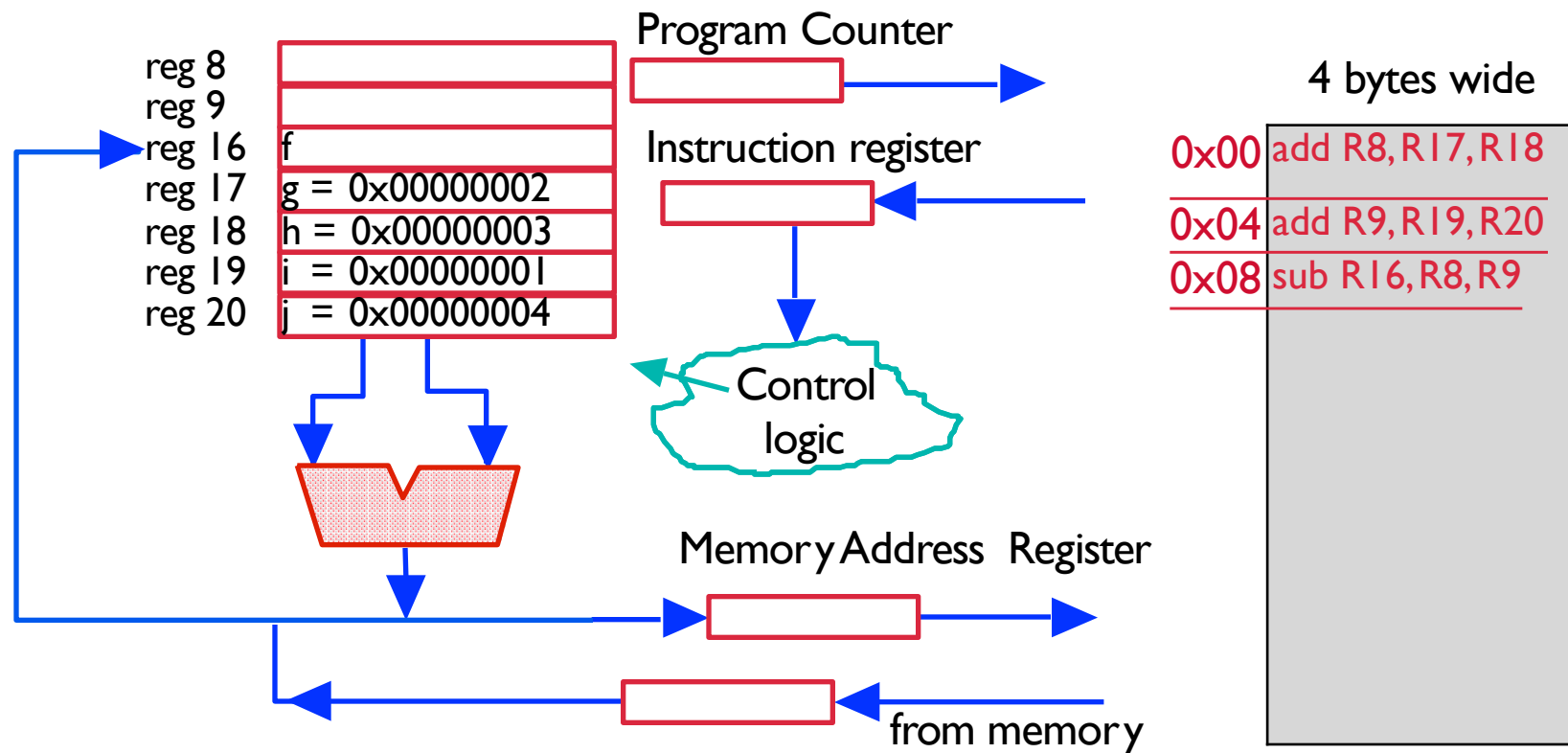
Execuție din tabela de registre



Alt exemplu

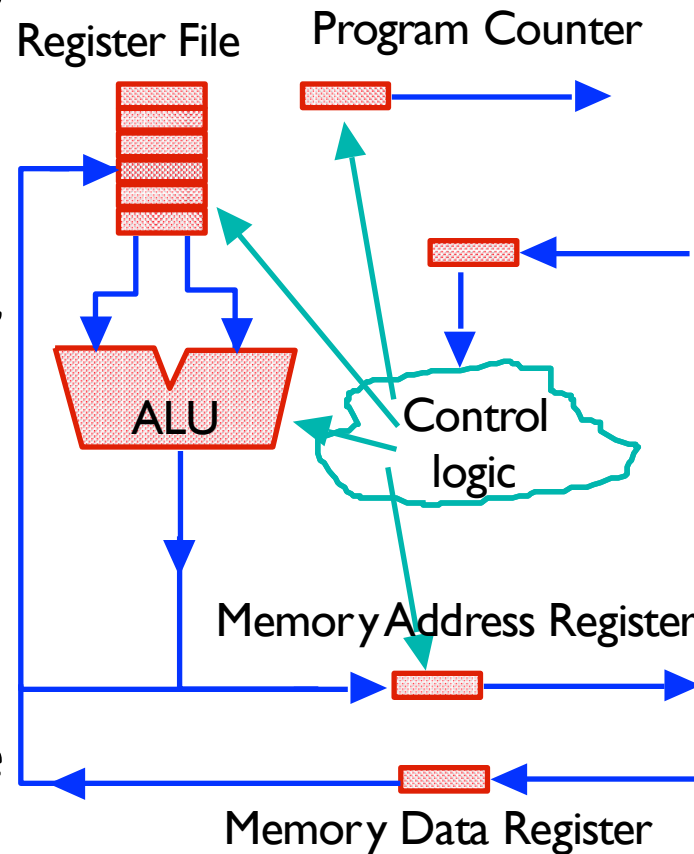
$$f = (g + h) - (i + j)$$

$$R16 == f, R17 == g, R18 == h, R19 == i, R20 == j$$



Accesarea datelor

- Adresa generată de ALU
- Adresa ajunge în *Memory Address Register*
- După accesul memoriei, rezultatul este întors în *Memory Data Register*
- Adresele pot fi pentru date sau pentru instrucțiuni, ambele sunt stocate în memorie

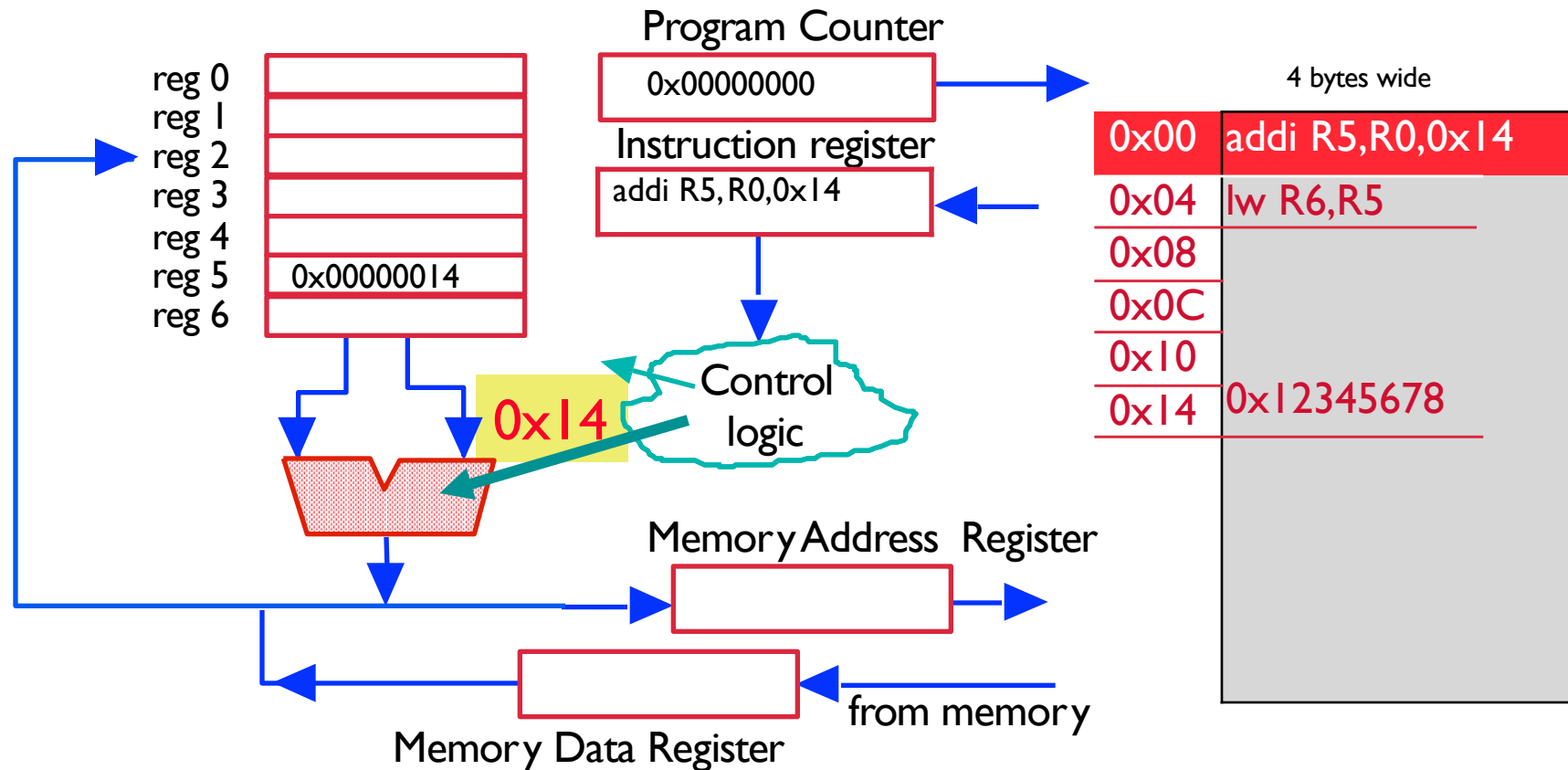


0x00	00101101
0x01	00100001
0x02	00110000
0x03	00001111
0x04	11010101
0x05	01010101
0x06	00101010
0x07	01010101
0x08	11110011
0x09	00111100
0x0A	00001100
0x0B	00000000
0x0C	00011000
0x0D	11111111

Operații cu memoria - Load

$R6 \leftarrow \text{mem}[0x14]$
Presupunem $\&A = 0x14$

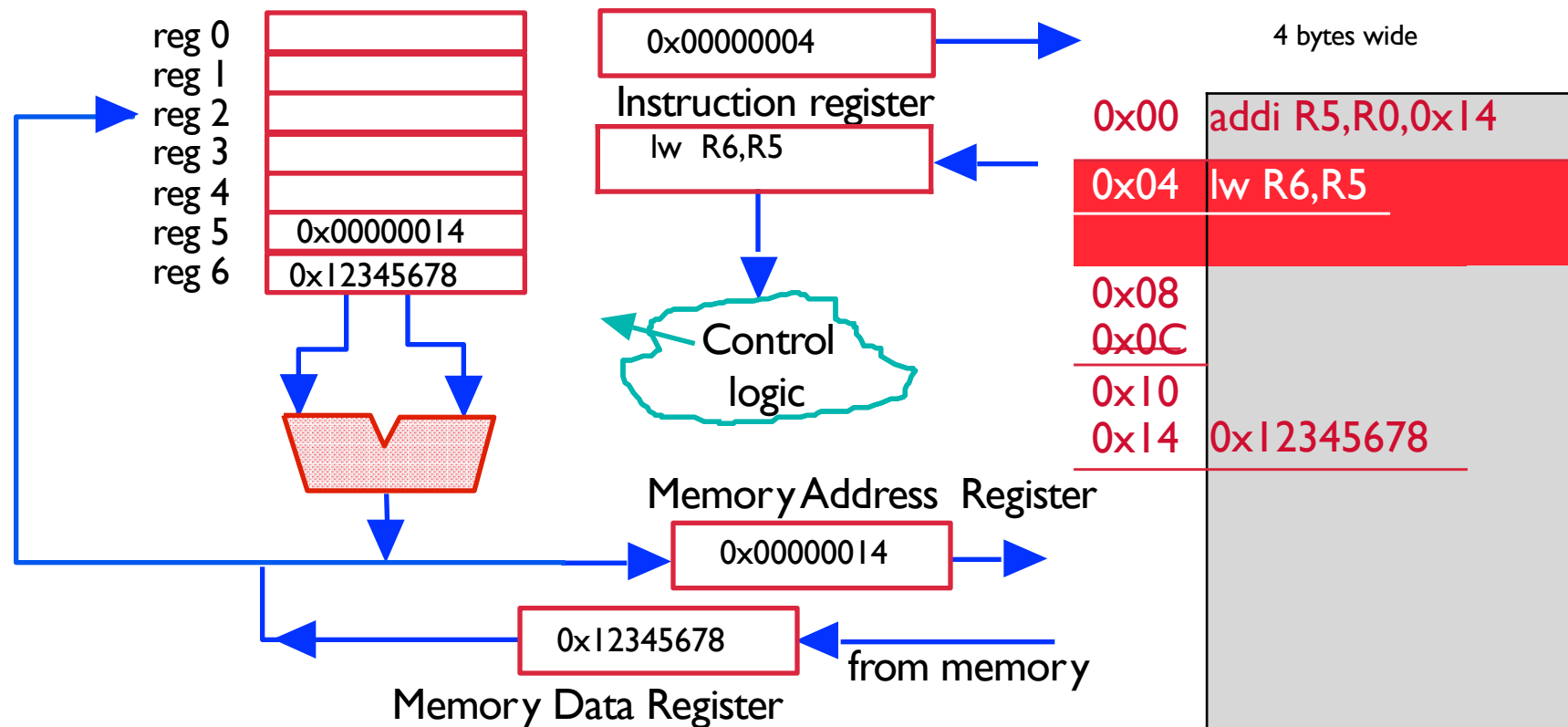
addi: Adună constanta pe 16 biți
la registrul sursă



Operații cu memoria - Load

$R6 \leftarrow \text{mem}[0x14]$

Presupunem $\&A = 0x14$

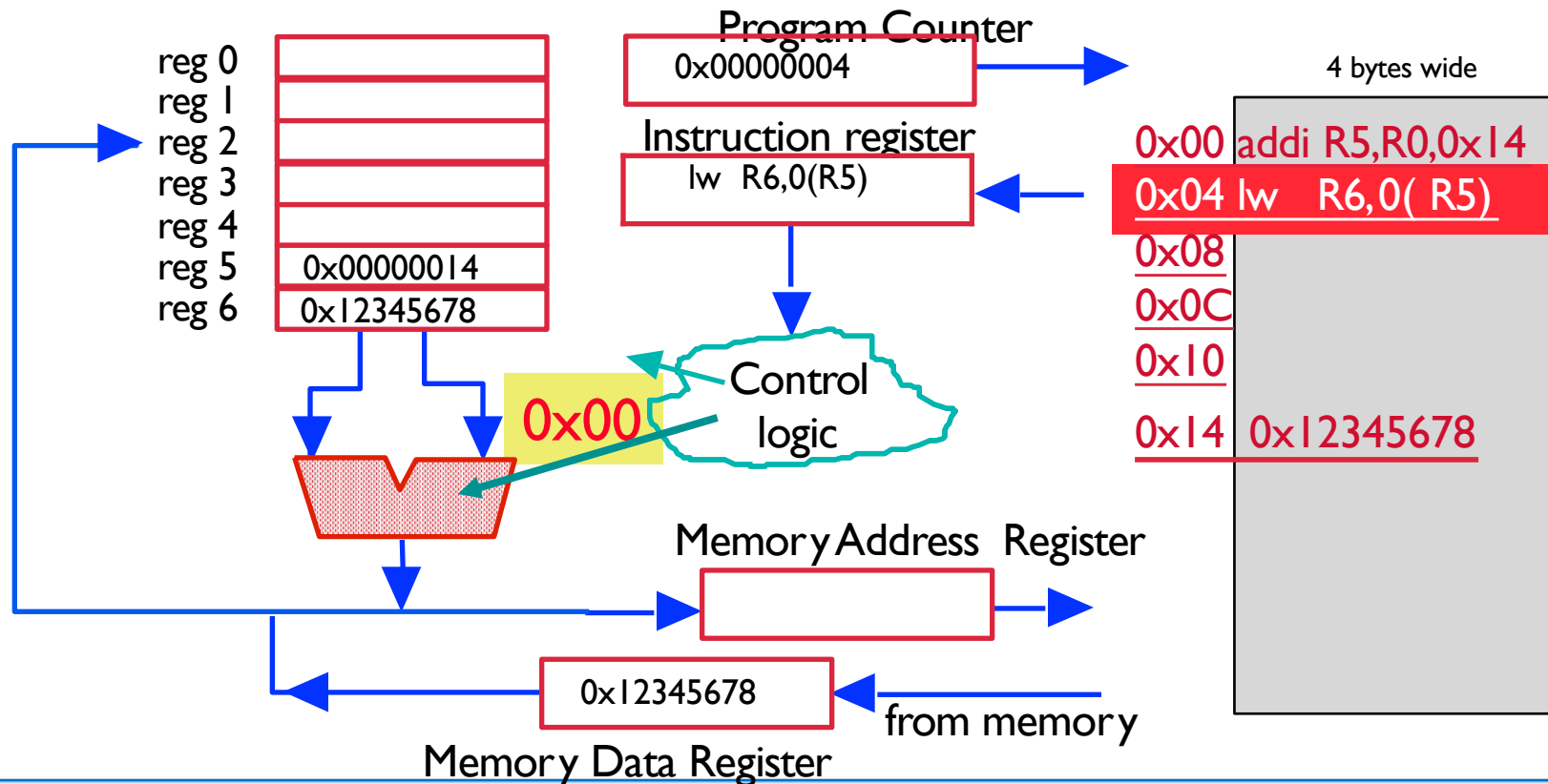


Operații cu memoria - Load

Adresa poate fi calculată și prin adăugarea unui **offset** registrului

LW R6,0(R5)

R6 ← memory[0 + R5]

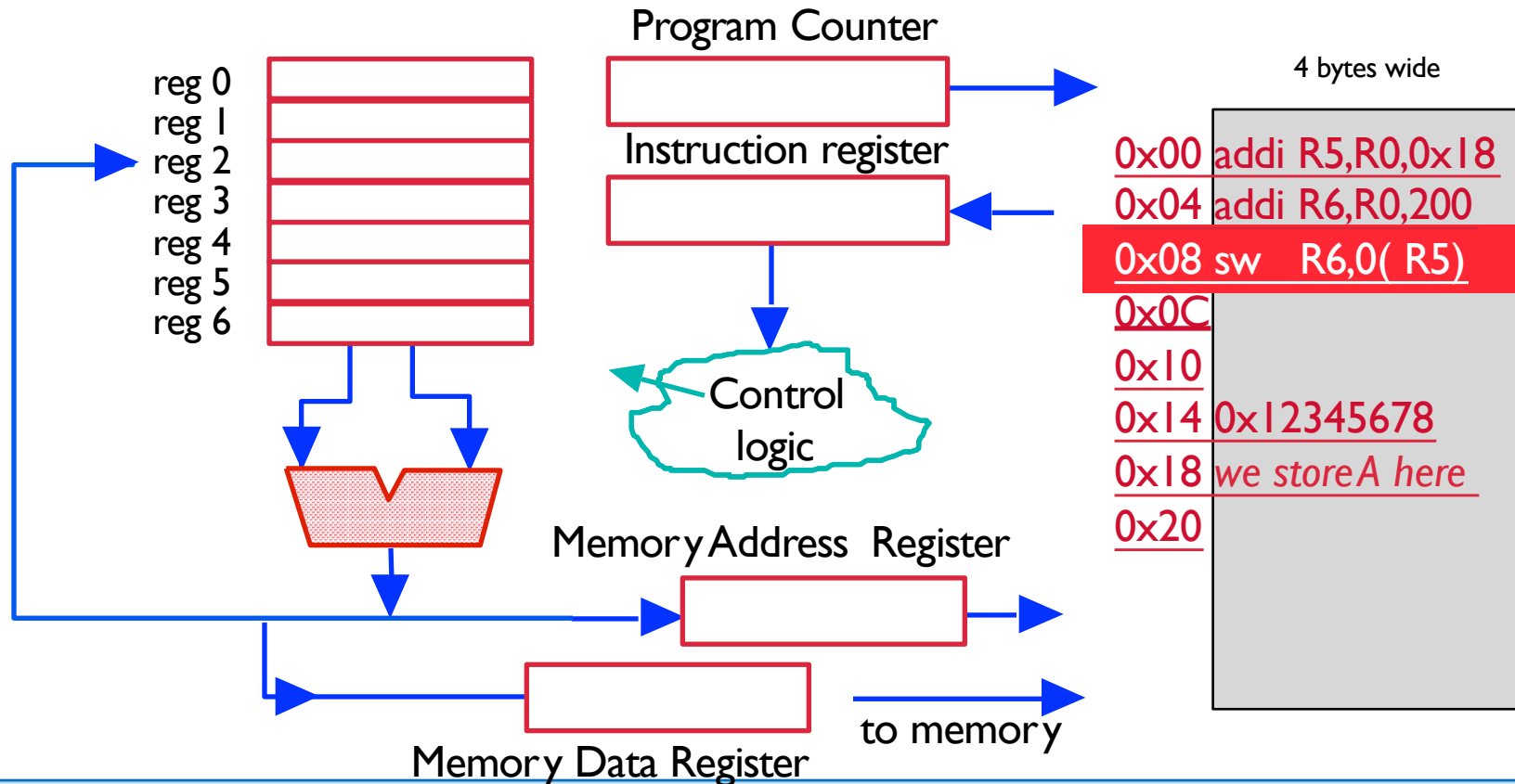


Operații cu memoria - Store

Stocarea datelor în memorie se face la fel

$A = 200$; presupunem $\&A = 0x18$

$\text{mem}[0x18] \leftarrow 200$



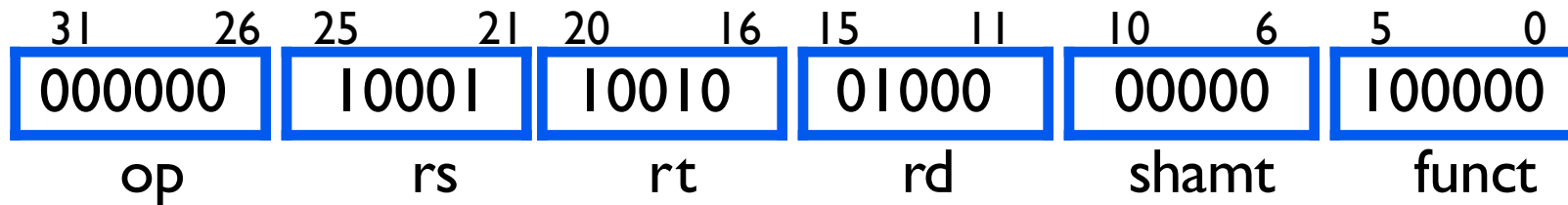
Formatul instrucțiunilor (cod mașină)

add R8, R17, R18

Stocat în format binar ca

0000010 00110010 01000000 00100000

Instrucțiunea este împărțită în "câmpuri"



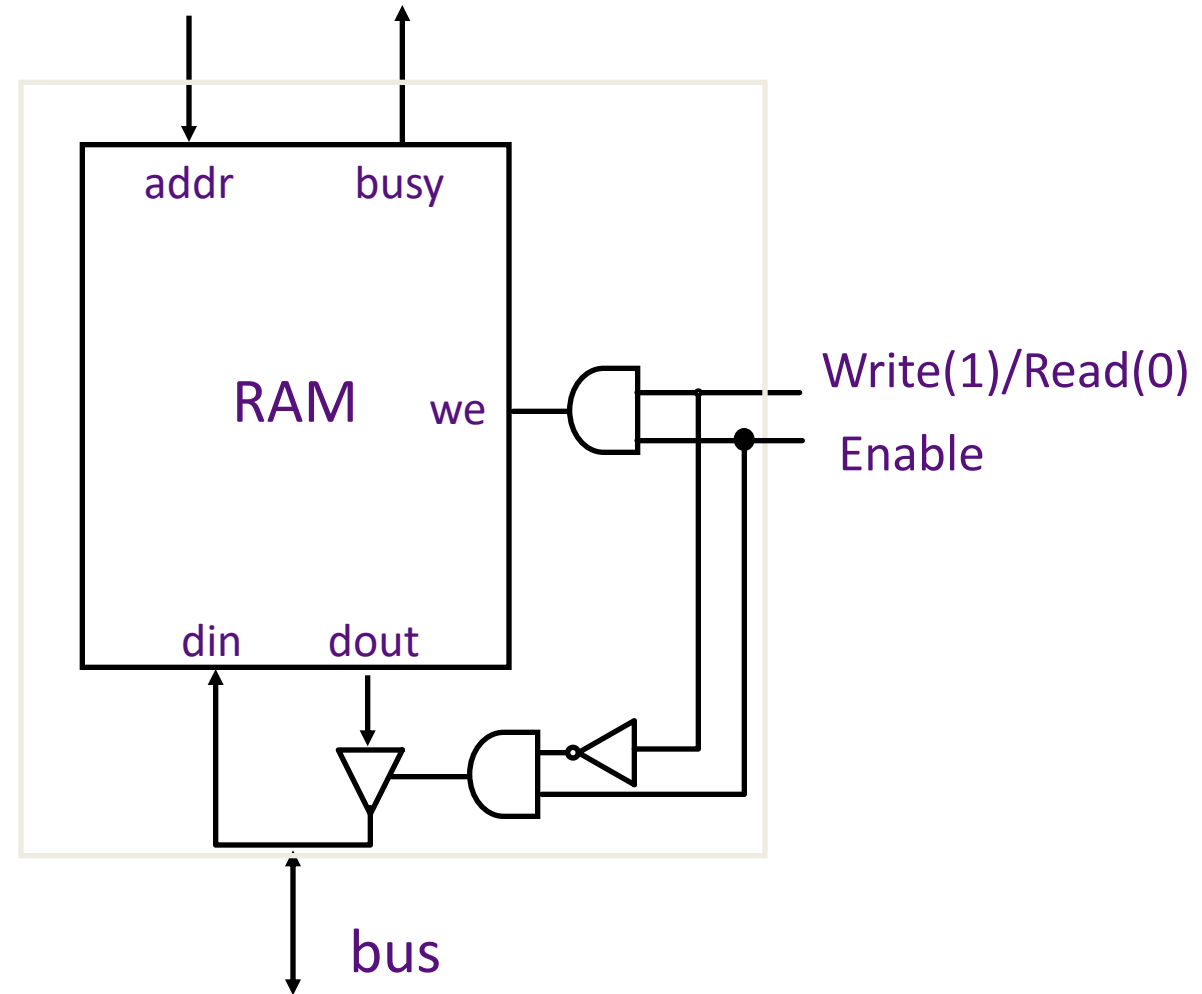
- op Codul operației
- rs Adresa registru sursă 1
- rt Adresa registru sursă 2
- rd Adresa registrul destinație
- shamt shift amount
- funct Funcție (selectează tipul operației)

add = 32_{10}

sub = 34_{10}

Modulul de memorie

Presupunere: memoria operează independent și e mult mai lentă decât operațiile de transfer registru-registru (mai mulți cicli de ceas pentru a accesa datele)



Execuția instrucțiunilor

Execuția unei instrucțiuni RISC-V presupune următorii pași:

1. instruction fetch
2. decode & register fetch
3. operații în UAL
4. operații cu memoria (opțional)
5. write back în tabela de registre generale (opțional)
+ calculul adresei următoarei instrucțiuni

Fragmente de microprogram

instr fetch:
MA, A \leftarrow PC
PC \leftarrow A + 4
IR \leftarrow Memory
dispatch on Opcode

*Poate fi văzută ca
O macroinstrucțiune*

ALU:
A \leftarrow Reg[rs1]
B \leftarrow Reg[rs2]
Reg[rd] \leftarrow func(A,B)
do instruction fetch

ALUi:
A \leftarrow Reg[rs1]
B \leftarrow Imm
Reg[rd] \leftarrow Opcode(A,B)
do instruction fetch

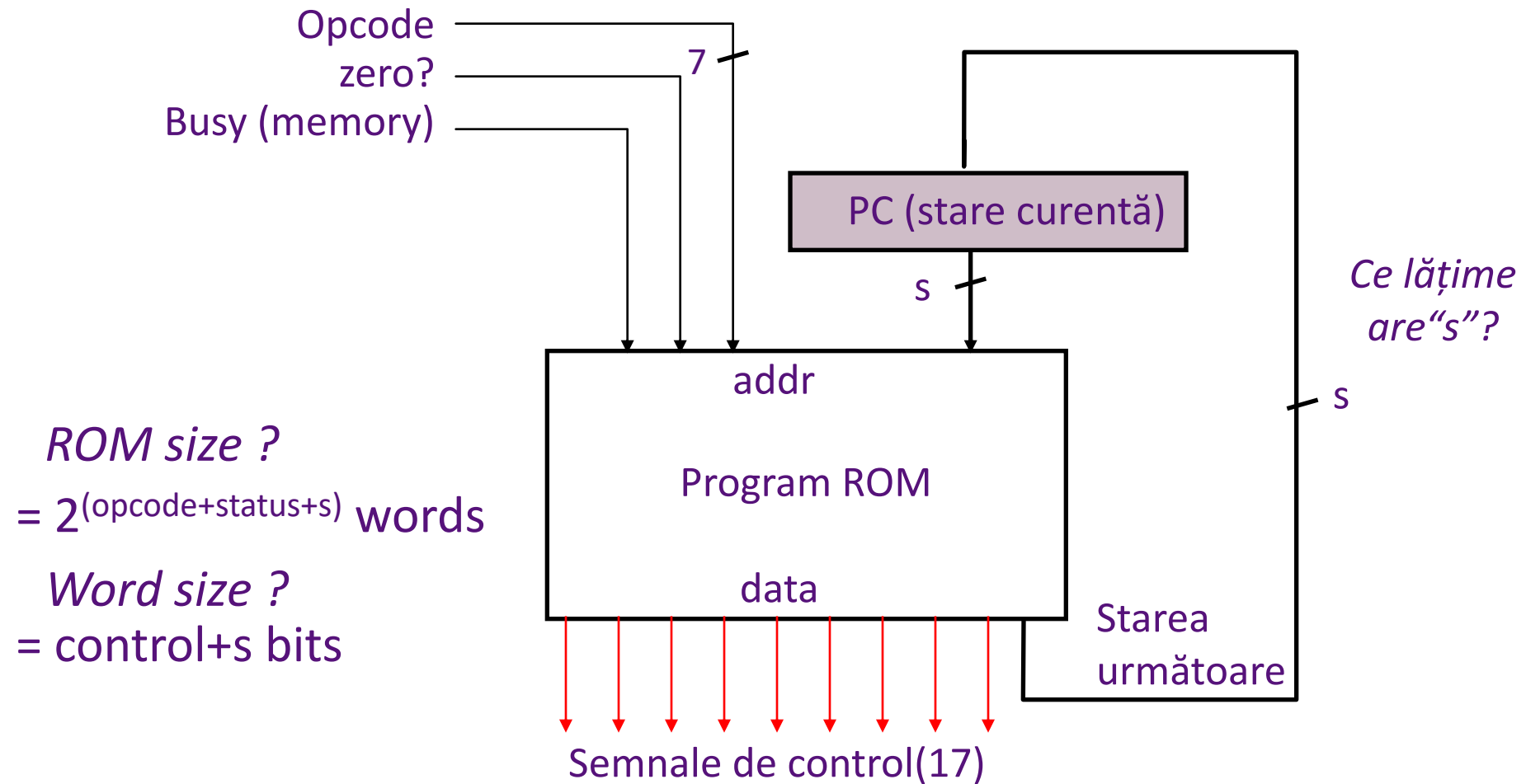
sign extension

Fragmente de Microprogram (*cont.*)

LW:	A <- Reg[rs1] B <- Imm MA <- A + B Reg[rd] <- Memory <i>do instruction fetch</i>
J:	A <- A - 4 Get original PC back in A B <- IR PC <- JumpTarg(A,B) <i>do instruction fetch</i>
beq:	A <- Reg[rs1] B <- Reg[rs2] <i>If A==B then go to bz-taken</i> <i>do instruction fetch</i>
bz-taken:	A <- PC A <- A - 4 Get original PC back in A B <- BImm << 1 BImm = IR[31:27,16:10] PC <- A + B <i>do instruction fetch</i>

$$\text{JumpTarg}(A,B) = \{A + (B[31:7] \ll 1)\}$$

RISC-V Microcontroller: *prima încercare: implementare numai în ROM*



Microprogramul din ROM

State	Op	zero?	busy	Control points	next-state
fetch ₀	*	*	*	MA, A ← PC	fetch ₁
fetch ₁	*	*	yes	fetch ₁
fetch ₁	*	*	no	IR ← Memory	fetch ₂
fetch ₂	*	*	*	PC ← A + 4	?
fetch ₂	ALU	*	*	PC ← A + 4	ALU ₀
ALU ₀	*	*	*	A ← Reg[rs1]	ALU ₁
ALU ₁	*	*	*	B ← Reg[rs2]	ALU ₂
ALU ₂	*	*	*	Reg[rd] ← func(A,B)	fetch ₀

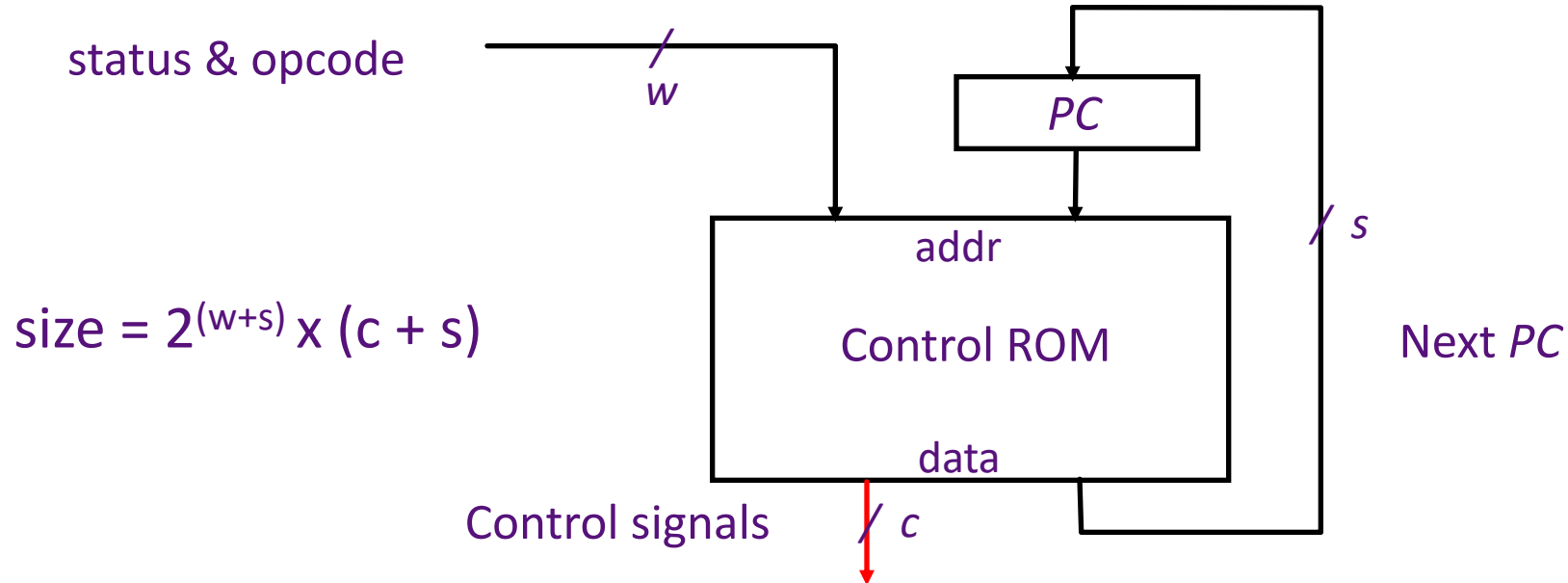
Microprogramul din ROM

State	Op	zero?	busy	Control points	next-state
fetch ₀	*	*	*	MA, A ← PC	fetch ₁
fetch ₁	*	*	yes	fetch ₁
fetch ₁	*	*	no	IR ⊛ Memory	fetch ₂
fetch ₂	ALU	*	*	PC ← A + 4	ALU ₀
fetch ₂	ALUi	*	*	PC ← A + 4	ALUi ₀
fetch ₂	LW	*	*	PC ← A + 4	LW ₀
fetch ₂	SW	*	*	PC ← A + 4	SW ₀
fetch ₂	J	*	*	PC ← A + 4	J ₀
fetch ₂	JAL	*	*	PC ← A + 4	JAL ₀
fetch ₂	JR	*	*	PC ← A + 4	JR ₀
fetch ₂	JALR	*	*	PC ← A + 4	JALR ₀
fetch ₂	beq	*	*	PC ← A + 4	beq ₀
...					
ALU ₀	*	*	*	A ← Reg[rs1]	ALU ₁
ALU ₁	*	*	*	B ← Reg[rs2]	ALU ₂
ALU ₂	*	*	*	Reg[rd] ← func(A,B)	fetch ₀

Microprogramul din ROM *Cont.*

State	Op	zero?	busy	Control points	next-state
ALUi ₀	*	*	*	A ← Reg[rs1]	ALUi ₁
ALUi ₁	*	*	*	B ← Imm	ALUi ₂
ALUi ₂	*	*	*	Reg[rd] ← Op(A,B)fetch ₀	
...					
J ₀	*	*	*	A ← A - 4	J ₁
J ₁	*	*	*	B ← IR	J ₂
J ₂	*	*	*	PC ← JumpTarg(A,B)	fetch ₀
...					
beq ₀	*	*	*	A ← Reg[rs1]	beq ₁
beq ₁	*	*	*	B ← Reg[rs2]	beq ₂
beq ₂	*	yes	*	A ← PC	beq ₃
beq ₂	*	no	*	fetch ₀
beq ₃	*	*	*	A ← A - 4	beq ₄
beq ₄	*	*	*	B ← BImm	beq ₅
beq ₅	*	*	*	PC ← A+B	fetch ₀

Dimensiunea memoriei ROM



$$\text{size} = 2^{(w+s)} \times (c + s)$$

RISC-V: $w = 5+2$ $c = 17$ $s = ?$

no. of steps per opcode = 4 to 6 + fetch-sequence

no. of states \rightarrow (4 steps per op-group) \times op-groups + secv. comune

$$= 4 \times 8 + 10 \text{ states} = 42 \text{ states} \Rightarrow s = 6$$

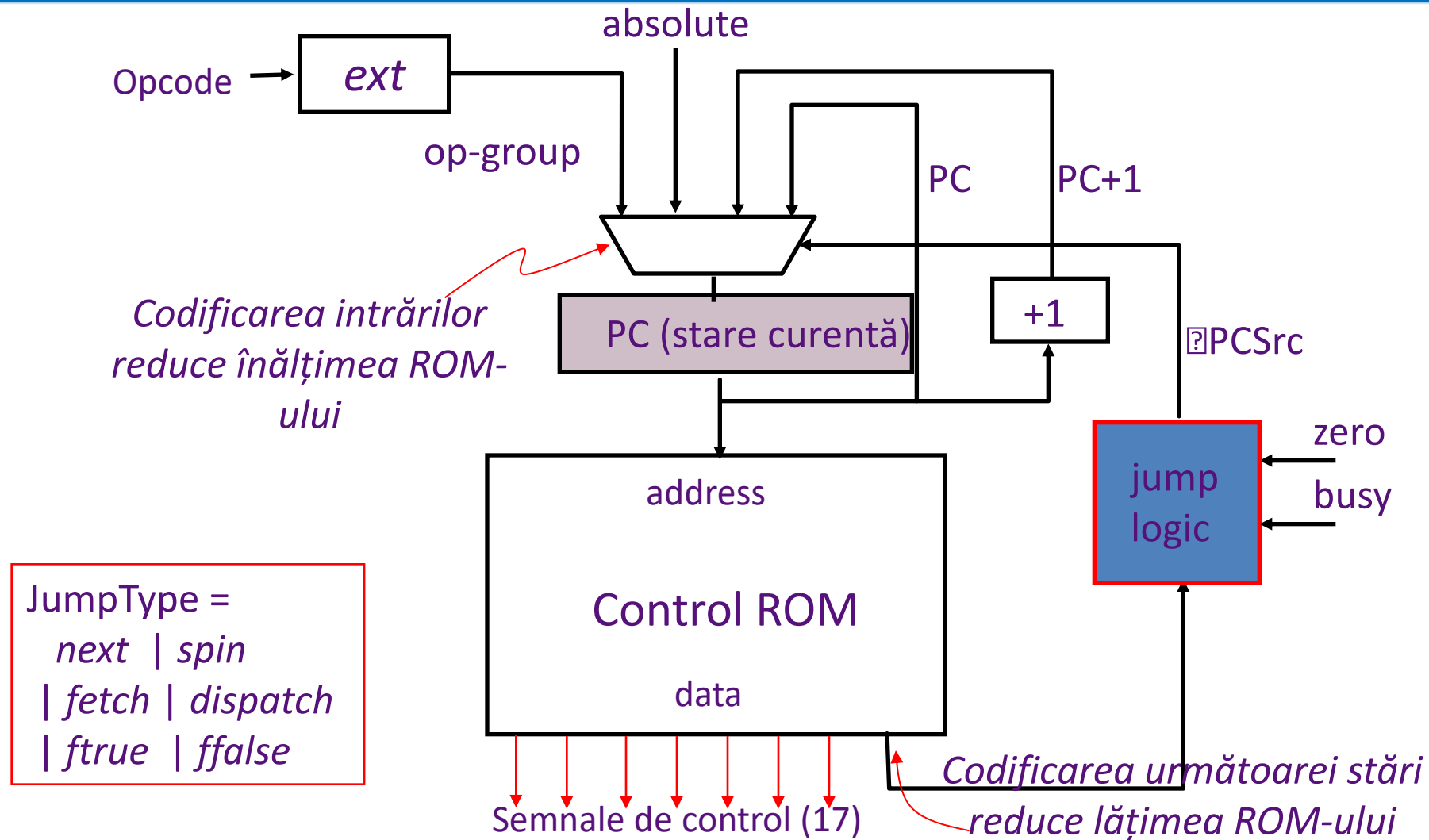
$$\text{Control ROM} = 2^{(5+6)} \times 23 \text{ bits} \sim 24 \text{ Kbytes}$$

Reducerea dimensiunii memoriei de control

Memoria de control trebuie să fie *rapidă => scumpă*

- Reducerea înălțimii ROM-ului (= biți de adresă)
 - *reducerea numărului de intrări prin adăugarea de logică externă*
Fiecare bit de intrare dublează dimensiunea memoriei de control
 - *reducerea numărului de stări prin gruparea codurilor de operații*
găsește secvențe comune de acțiuni
 - *comprimă biții de stare pentru intrări*
combină toate excepțiile într-una singură, i.e.,
exception/no-exception
- Reducerea lățimii ROM-ului
 - *restricționează codificarea următorii stări*
Next, Dispatch on opcode, Wait for memory, ...
 - *codifică semnalele de control (microcod verticalizat)*

RISC-V Controller V2



Stările pentru Jump

PCSrc = *Case* JumpTypes

next	->	PC+1
spin	->	if (busy) then PC else PC+1
fetch	->	absolute
dispatch	->	op-group
ftrue	->	if (zero) then absolute else PC+1
ffalse	->	if (zero) then PC+1 else absolute

Instruction Fetch & ALU: *RISC-V-Controller-2*

State	Control points	next-state
fetch ₀	MA, A ← PC	
fetch ₁	IR ← Memory	
fetch ₂	PC ← A + 4	
...		
ALU ₀	A ← Reg[rs1]	
ALU ₁	B ← Reg[rs2]	
ALU ₂	Reg[rd] ← func(A, B)	
ALUi ₀	A ← Reg[rs1]	
ALUi ₁	B ← Imm	
ALUi ₂	Reg[rd] ← Op(A, B)	

Load & Store: *RISC-V-Controller-2*

State	Control points	next-state
LW_0	$A \leftarrow \text{Reg}[\text{rs1}]$	next
LW_1	$B \leftarrow \text{Imm}$	next
LW_2	$MA \leftarrow A+B$	next
LW_3	$\text{Reg}[\text{rd}] \leftarrow \text{Memory}$	spin
LW_4		fetch
SW_0	$A \leftarrow \text{Reg}[\text{rs1}]$	next
SW_1	$B \leftarrow \text{BImm}$	next
SW_2	$MA \leftarrow A+B$	next
SW_3	$\text{Memory} \leftarrow \text{Reg}[\text{rs2}]$	spin
SW_4		fetch

Branches: *RISC-V-Controller-2*

State	Control points	next-state
beq ₀	A ← Reg[rs1]	next
beq ₁	B ← Reg[rs2]	next
beq ₂	A ← PC	ffalse
beq ₃	A ← A - 4	next
beq ₃	B ← BImm << 1	next
beq ₄	PC ← A + B	fetch

Jumps: *RISC-V-Controller-2*

State	Control points	next-state
J_0	$A \leftarrow A-4$	next
J_1	$B \leftarrow IR$	next
J_2	$PC \leftarrow \text{JumpTarg}(A,B)$	fetch
JR_0	$A \leftarrow \text{Reg}[rs1]$	next
JR_1	$PC \leftarrow A$	fetch
JAL_0	$A \leftarrow PC$	next
JAL_1	$\text{Reg}[1] \leftarrow A$	next
JAL_2	$A \leftarrow A-4$	next
JAL_3	$B \leftarrow IR$	next
JAL_4	$PC \leftarrow \text{JumpTarg}(A,B)$	fetch

VAX 11-780 Microcode

```

P1WFUD,1 [600,1205] MICRO2 1F(12) 26-May-81 14:58:11 VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122 Page 771
CALL2 ,mic [600,1205] Procedure call : CALLG, CALLS

;29744 ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
;29745
;29746 =0 ;-----;CALL SITE FOR MPUSH
;29747 CALL.7: D_Q,AND,RC[T2], ;STRIP MASK TO BITS 11-0
;29748 CALL,J/MPUSH ;PUSH REGISTERS

;29749
;29750 ;-----;RETURN FROM MPUSH
;29751 CACHE_D[LONG], ;PUSH PC
;29752 LAB_R[SP] ; BY SP

;29753
;29754 ;-----;
;29755 CALL.8: R[SP]&VA_LA-K[.8] ;UPDATE SP FOR PUSH OF PC &
;29756
;29757 ;-----;
;29758 D_R[FP] ;READY TO PUSH FRAME POINTER

;29759
;29760 =0 ;-----;CALL SITE FOR PSHSP
;29761 CACHE_D[LONG], ;STORE FP,
;29762 LAB_R[SP], ; GET SP AGAIN
;29763 SC_K[.FFF0], ;-16 TO SC
;29764 CALL,J/PSHSP

;29765
;29766 ;-----;
;29767 D_R[AP], ;READY TO PUSH AP
;29768 Q_ID[PSL] ; AND GET PSW FOR COMBINATIO

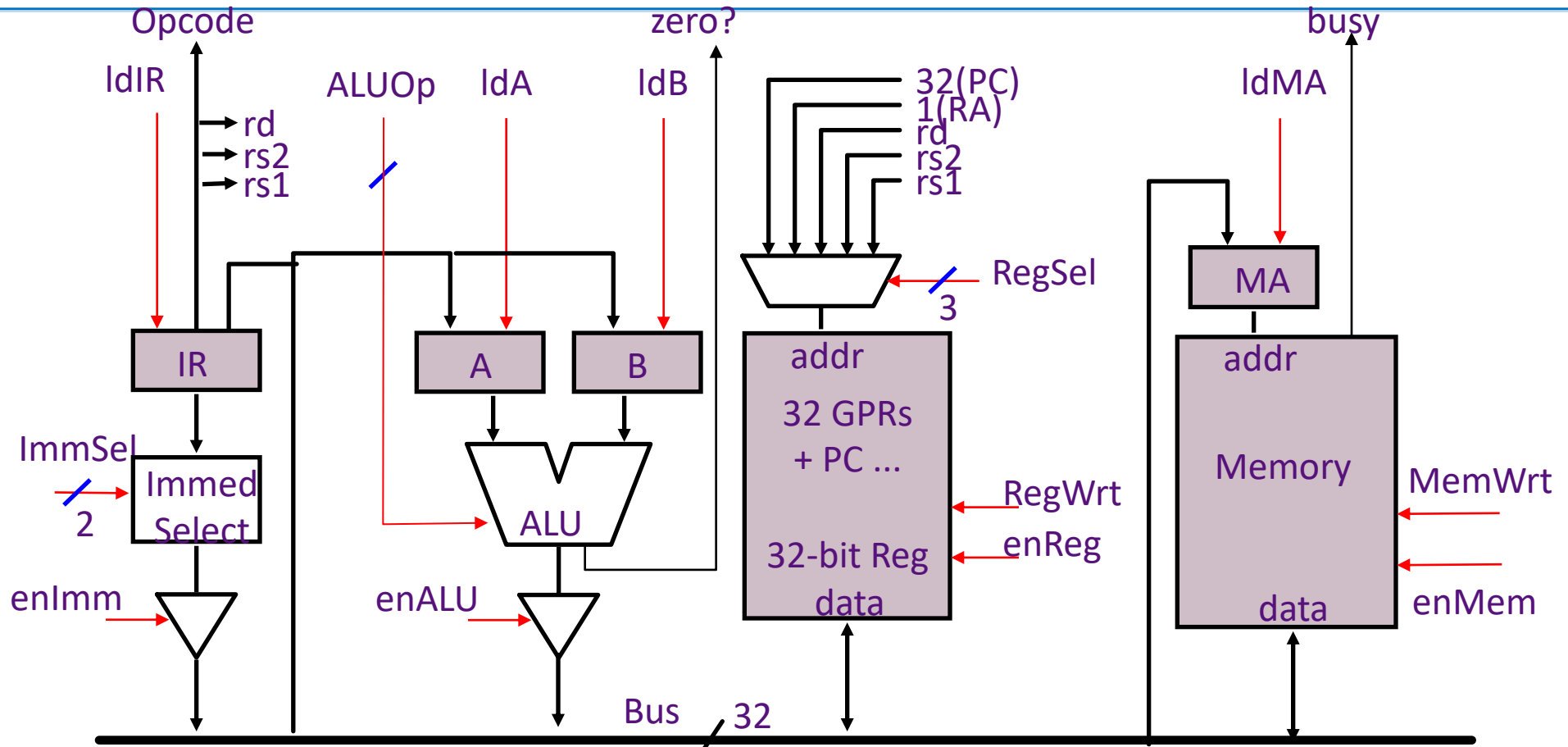
;29769
;29770 ;-----;
;29771 CACHE_D[LONG], ;STORE OLD AP
;29772 Q_Q,ANDNOT,K[.1F], ;CLEAR PSW<T,N,Z,V,C>
;29773 LAB_R[SP] ;GET SP INTO LATCHES AGAIN

;29774
;29775 ;-----;
;29776 PC&VA_RC[T1], FLUSH.IB ; LOAD NEW PC AND CLEAR OUT

;29777
;29778 ;-----;
;29779 D_DAL,SC, ;PSW TO D<31:16>
;29780 Q_RC[T2], ;RECOVER MASK
;29781 SC_SC+K[.3], ;PUT -13 IN SC
;29782 LOAD.IB, PC_PC+1 ;START FETCHING SUBROUTINE I

;29783
;29784 ;-----;
;29785 D_DAL,SC, ;MASK AND PSW IN D<31:03>
;29786 Q_PC[T4], ;GET LOW BITS OF OLD SP TO Q<1:0>
;29787 SC_SC+K[.A] ;PUT -3 IN SC
;29788

```

$rd \leftarrow M[(rs1)] \text{ op } (rs2)$
 $M[(rd)] \leftarrow (rs1) \text{ op } (rs2)$
 $M[(rd)] \leftarrow M[(rs1)] \text{ op } M[(rs2)]$

Reg-Memory-src ALU op
Reg-Memory-dst ALU op
Mem-Mem ALU op

Instrucțiuni Mem-Mem ALU:

RISC-V-Controller-2

Mem-Mem ALU op $M[(rd)] \leftarrow M[(rs1)] \text{ op } M[(rs2)]$

ALUMM ₀	MA \leftarrow Reg[rs1]	next
ALUMM ₁	A \leftarrow Memory	spin
ALUMM ₂	MA \leftarrow Reg[rs2]	next
ALUMM ₃	B \leftarrow Memory	spin
ALUMM ₄	MA \leftarrow Reg[rd]	next
ALUMM ₅	Memory \leftarrow func(A,B)	spin
ALUMM ₆		fetch

Instrucțiunile complexe nu necesită de obicei modificări ale structurii unității de control microprogramat

-- doar spațiu suplimentar în memoria ROM de control

Implementarea acestor instrucțiuni folosind un controller hardware dedicat nu poate fi făcută fără modificarea căilor de date din interiorul unității de control.

Considerente de performanță

Unitatea de control microprogramat
-> cicli multipli per instrucțiune

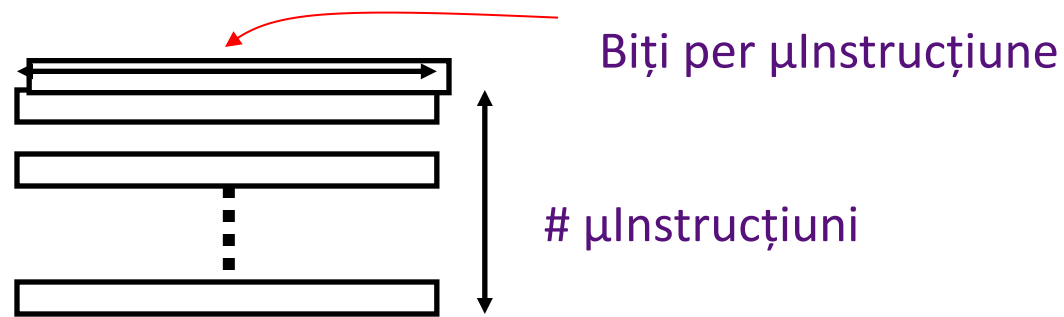
Durata ciclului ?

$$t_C > \max(t_{\text{reg-reg}}, t_{\text{ALU}}, t_{\text{ROM}})$$

Presupunem că $10 * t_{\text{ROM}} < t_{\text{RAM}}$

Performanță mărită, atunci când comparăm cu o implementare pur hardware a UC care execută toate operațiile într-un singur ciclu.

μCod orizontal vs. vertical



- μcodul orizontalizat are μinstrucțiuni mai lungi
 - Mai multe operații per μinstrucțiune
 - Mai puțini pași în microcod pe macroinstrucțiune
 - Sparse encoding ⇒ mai mulți biți
- μcodul verticalizat are μinstrucțiuni scurte
 - De obicei, o singură operație de transfer date per μinstrucțiune
 - μinstrucțiuni separate pentru branch
 - Mai mulți pași în microcod per macroinstrucțiune
 - Mai compact ⇒ mai puțini biți
- Nanocoding
 - Încearcă să combine cele două practici

Nanocoding

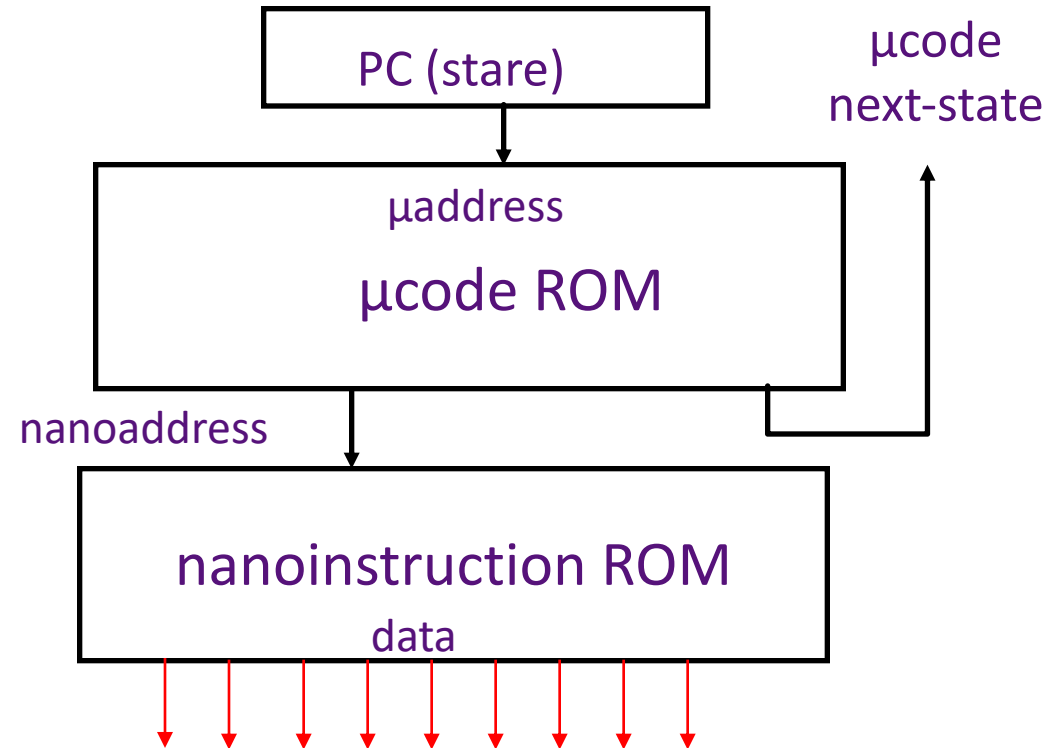
Exploatează existența
secvențelor recurente de
μcod, e.g.,

ALU₀ A ← Reg[rs1]

...

ALUi₀ A ← Reg[rs1]

...



MC68000 avea un microcod pe 17-biți ce conținea un μjump pe 10 biți sau un pointer pe 9 biți la o nanoinstrucțiune

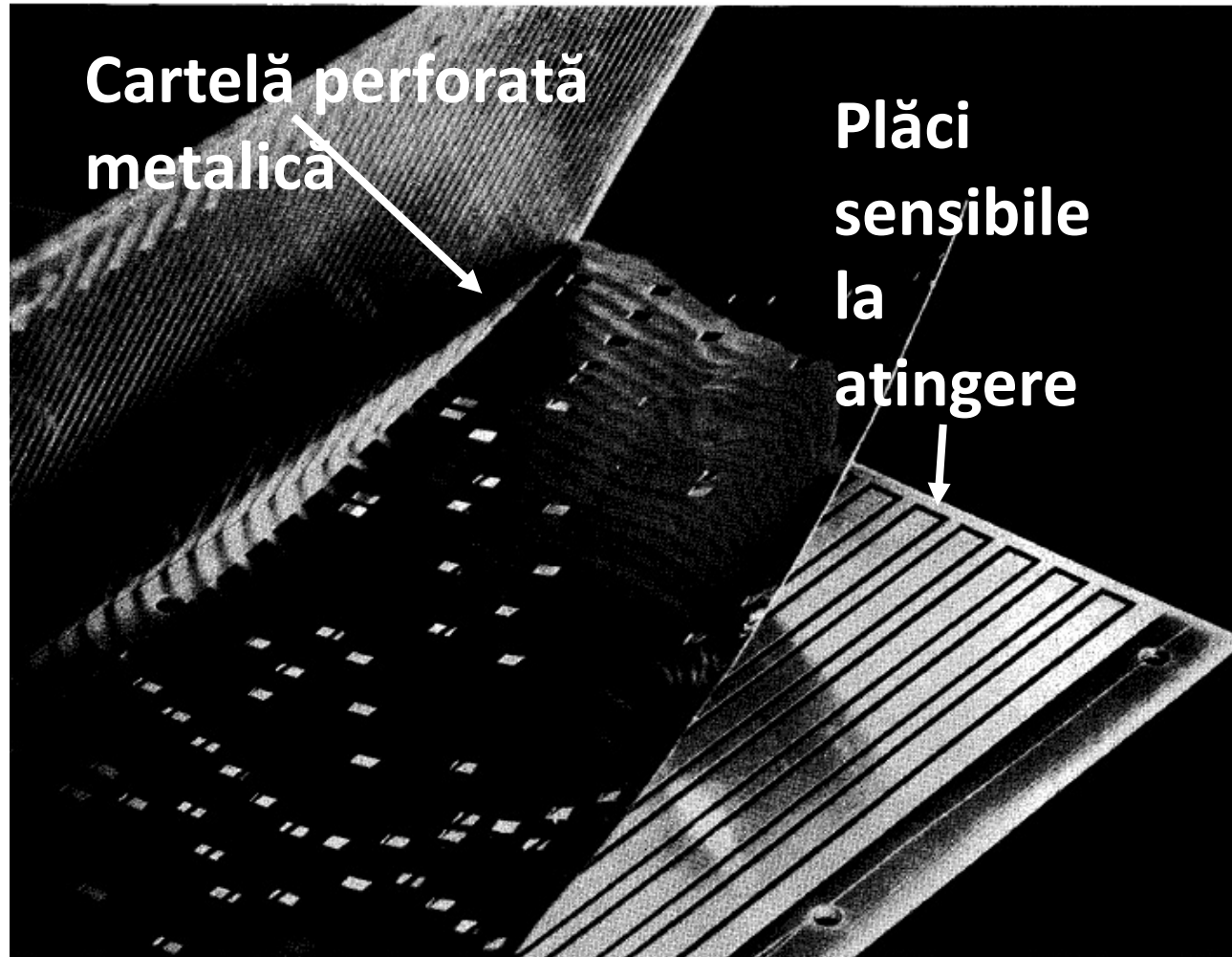
- Nanoinstrucțiunile aveau 68 de biți lățime și decodificate produceau 196 semnale de control

Microprogramarea pentru IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
μ inst width (bits)	50	52	85	87
μ code size (K μ insts)	4	4	2.75	2.75
μ store technology	CCROS	TCROS	BCROS	BCROS
μ store cycle (ns)	750	625	500	200
memory cycle (ns)	1500	2500	2000	750
Rental fee (\$K/month)	4	7	15	35

Numai cele mai rapide modele (75 și 95) aveau UC hardwired

IBM Card Capacitor Read-Only Storage



Cartelă perforată
metalică

Plăci
sensibile
la
atingere

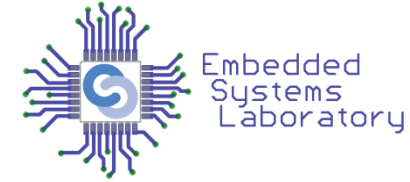
[IBM Journal, January 1961]

Microprogramarea a atins apogeul în anii '70

- Memoriile ROM erau cu mult mai rapide decât cele DRAM
- Pentru un ISA complex, unitatea de control microprogramat era mai ieftină și mai simplu de fabricat
- *Instrucțiunile noi*, e.g., floating point, puteau fi adăugate fără modificări ale magistralelor de date
- *Repararea defectelor* controllerului era mai simplă
- Compatibilitatea ISA a numeroaselor modele de calculatoare putea fi obținută ușor și ieftin

Aproape toate calculatoarele acelei epoci aveau o unitate de control microprogramată

Writable Control Store (WCS)



- Implementăm memoria de control în RAM, nu în ROM
 - Memoriile MOS SRAM sunt acum aproape la fel de rapide ca un ROM (memoriile cu mieș de ferită/DRAM erau de 2-10x mai lente)
 - E foarte greu să scrii un microprogram fără erori din prima încercare
- Opțiunea de User-WCS pentru anumite procesoare
 - Permite utilizatorilor să schimbe microcodul pentru procesorul în cauză
- User-WCS *a eșuat*
 - Nu au existat suficiente programe care să folosească această abilitate
 - Dificil de stocat software util într-un spațiu atât de mic
 - Unitatea de control era optimizată pentru microcodul original, mai puțin pentru restul
 - Protecția integrității ISA dificilă dacă utilizatorul poate schimba microcodul
 - Memoria virtuală necesita microcod restartabil

Microprogramarea nu a dispărut

- A jucat un rol crucial în dezvoltarea microprocesoarelor anilor 80
 - DEC uVAX, Motorola 68K series, Intel 286/386
- Joacă un rol secundar, dar important, în microprocesoarele moderne
 - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, ...
 - Majoritatea instrucțiunilor sunt executate direct (logică cablată)
 - Instrucțiunile utilizate mai puțin frecvent și/sau complicate, folosesc microcod
- Se pot aplica patch-uri la microcod pentru a repara anumite erori ISA detectate post-fabricare (ex. Procesoarele Intel încarcă patch-uri de microcod la bootup)

Acknowledgements

- Aceste slide-uri conțin materiale aparținând:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
 - Behrooz Parhami (UCSB)
- MIT material derived from course 6.823
- UCB material derived from course CS252