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/

Up to

24GB

LPDDR5 memory

High-performance media engine

40%

Faster Neural Engine

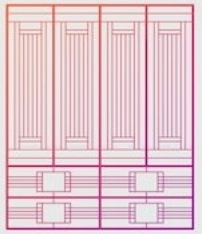
Up to

15.8 trillion

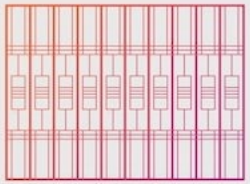
operations per second

16-core Neural Engine

Second-generation 5 nm technology



8-core CPU



Up to

10-core GPU

ProRes

encode and decode

6K external display support

18%

Faster CPU

35%

Faster GPU

Over

20 billion

transistors

Industry-leading performance per watt

50%

More memory bandwidth

100GB/s

Memory bandwidth

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. *microarchitectură*)
- Mai multe implementări posibile pentru aceeași ISA
  - E.g., sovieticii au construit clone compatibile ale IBM360, precum si 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, Apple, etc.

Maparea ISA pe microarhitectură

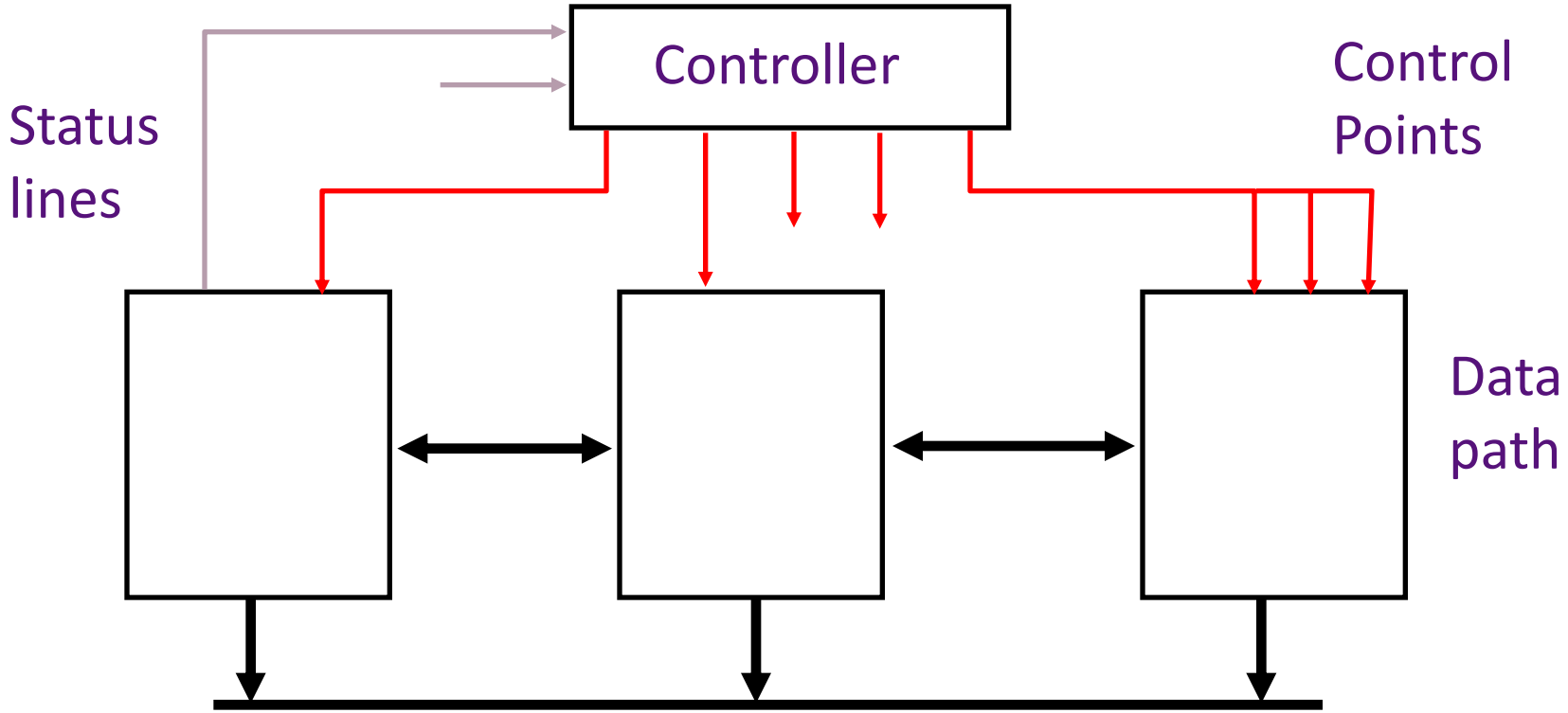
- ISA poate fi proiectată ținând cont tipul de microarhitectură:
  - Acumulator ⇒ hardwired, unpipelined
  - CISC ⇒ microcoded
  - RISC ⇒ hardwired, pipelined
  - VLIW ⇒ fixed-latency in-order parallel pipelines
  - JVM ⇒ software interpretation
- Dar poate fi implementată folosind orice fel de microarhitectură
  - 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.

Microarchitectură: *Implementarea unei ISA*



Structură: Cum sunt interconectate componentele.

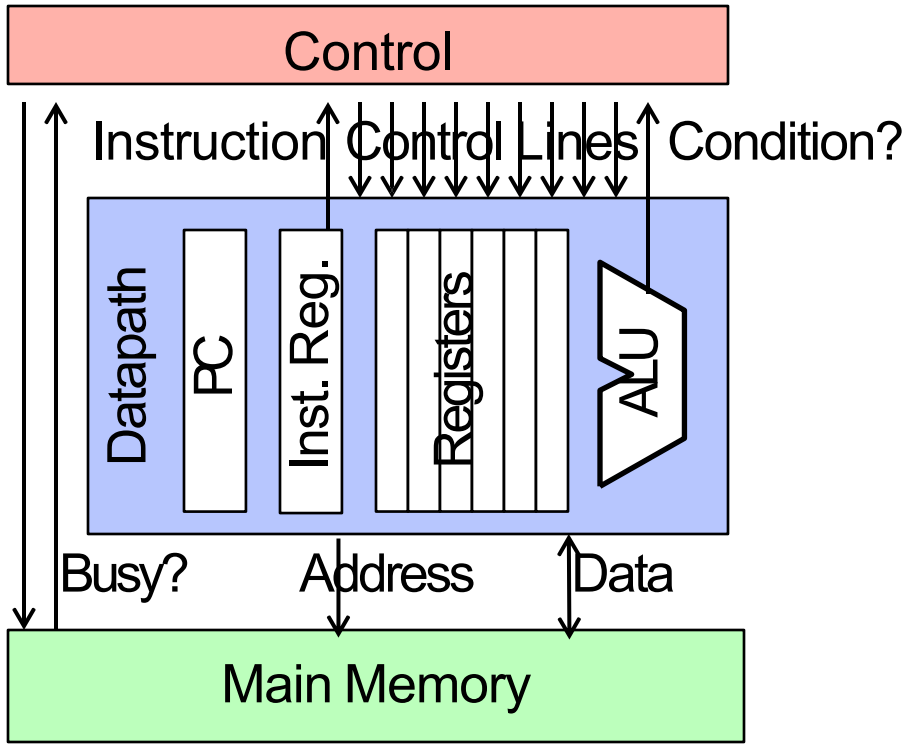
Statică

Comportament: Cum se miscă datele între diferitele componente

Dinamică

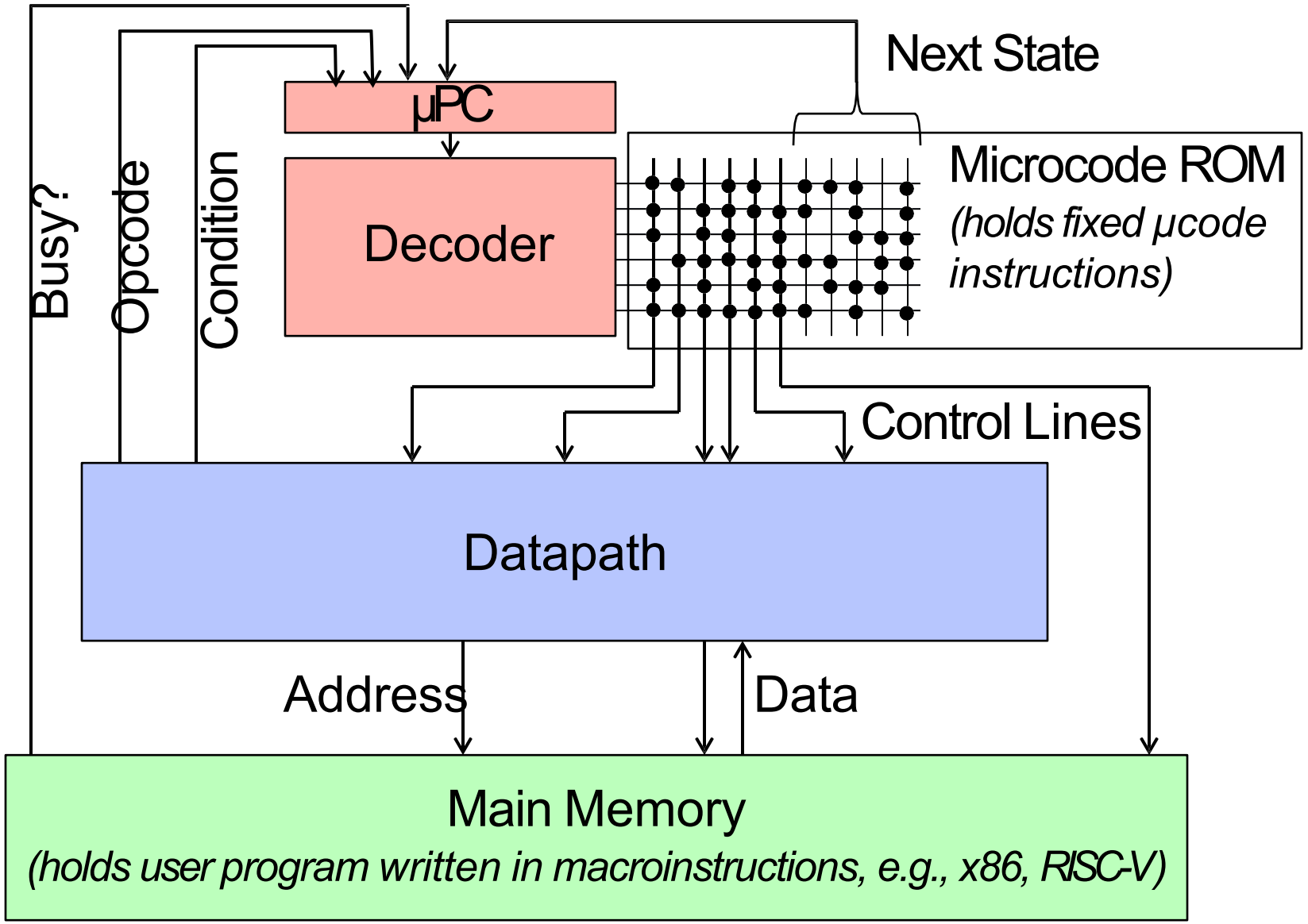
Control vs. Datapath

Structura unui procesor poate fi împărțită în calea de *date* (datapath) unde numerele sunt stocate și sunt executate operațiile aritmetice și logice și calea de *control*, care secvențiază execuția operațiilor pe calea de date

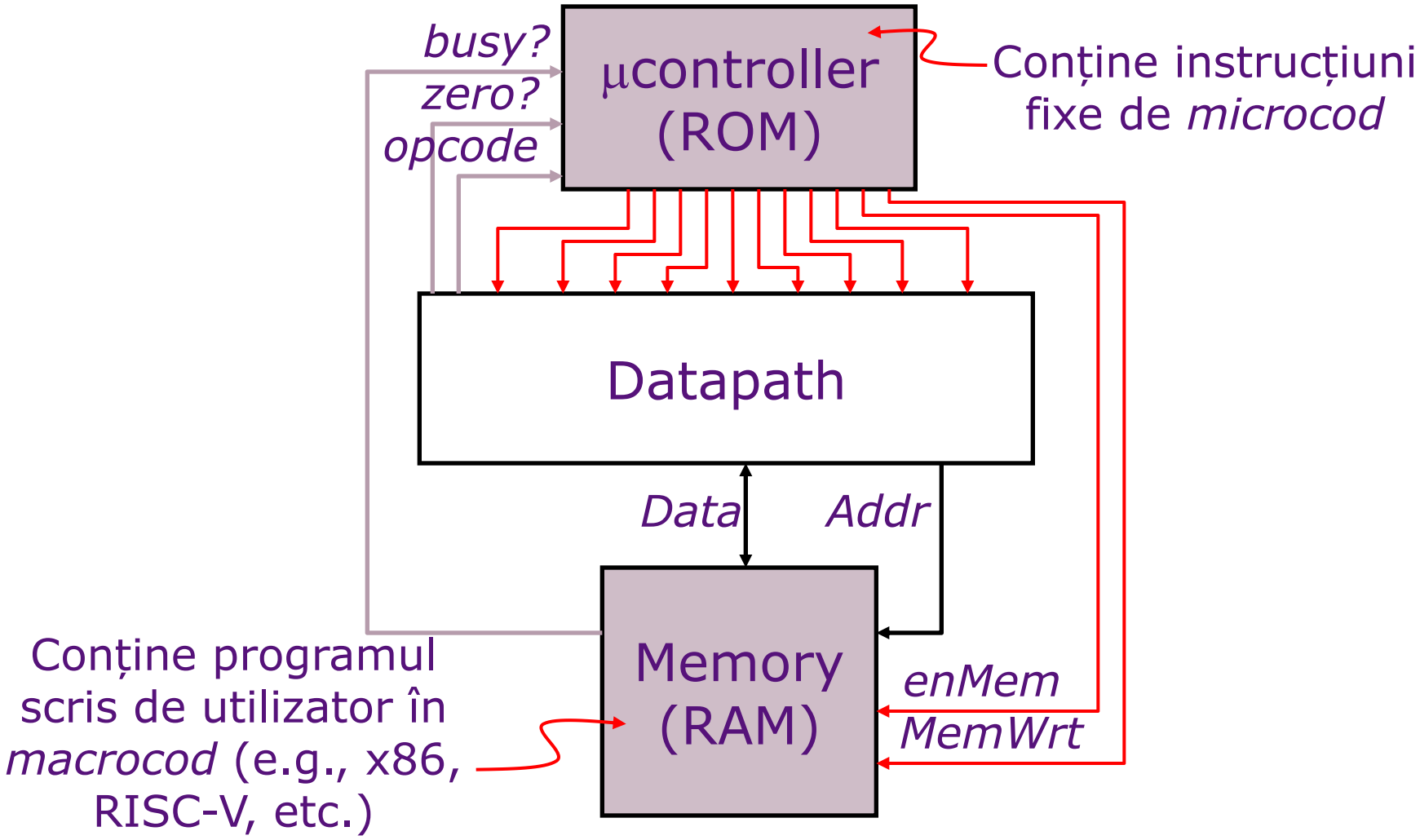


- O provocare mare în designul primelor calculatoare era cum să proiectezi corect circuitele de control
- Maurice Wilkes a inventat ideea de microprogramare pentru a proiecta unitatea de control a lui EDSAC-II în 1958

Microcoded CPU



Microarchitectură 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

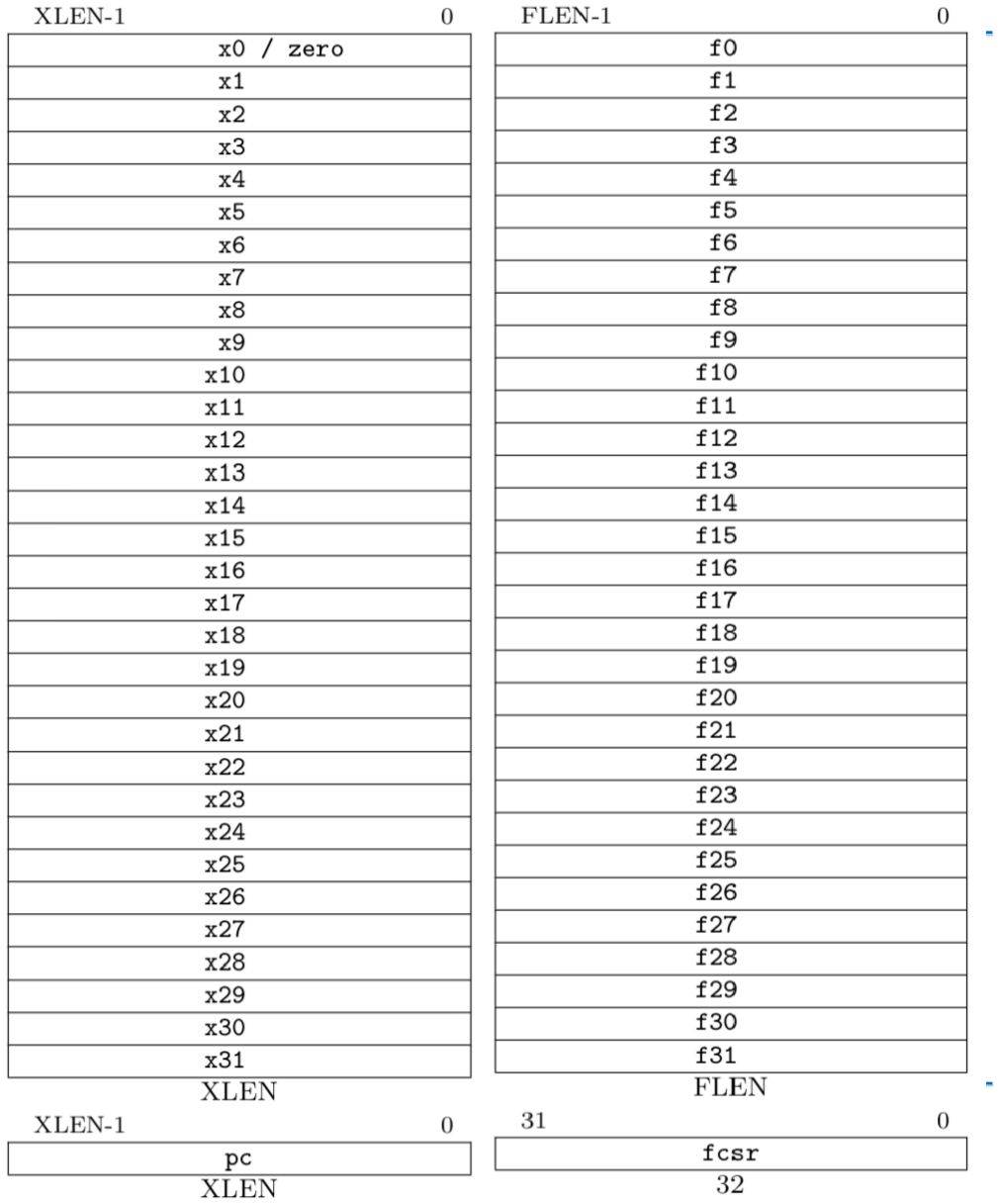
RV32 Processor State

Program counter (**pc**)

32x32-bit registre pt întregi (**x0-x31**)
- **x0** conține întotdeauna 0

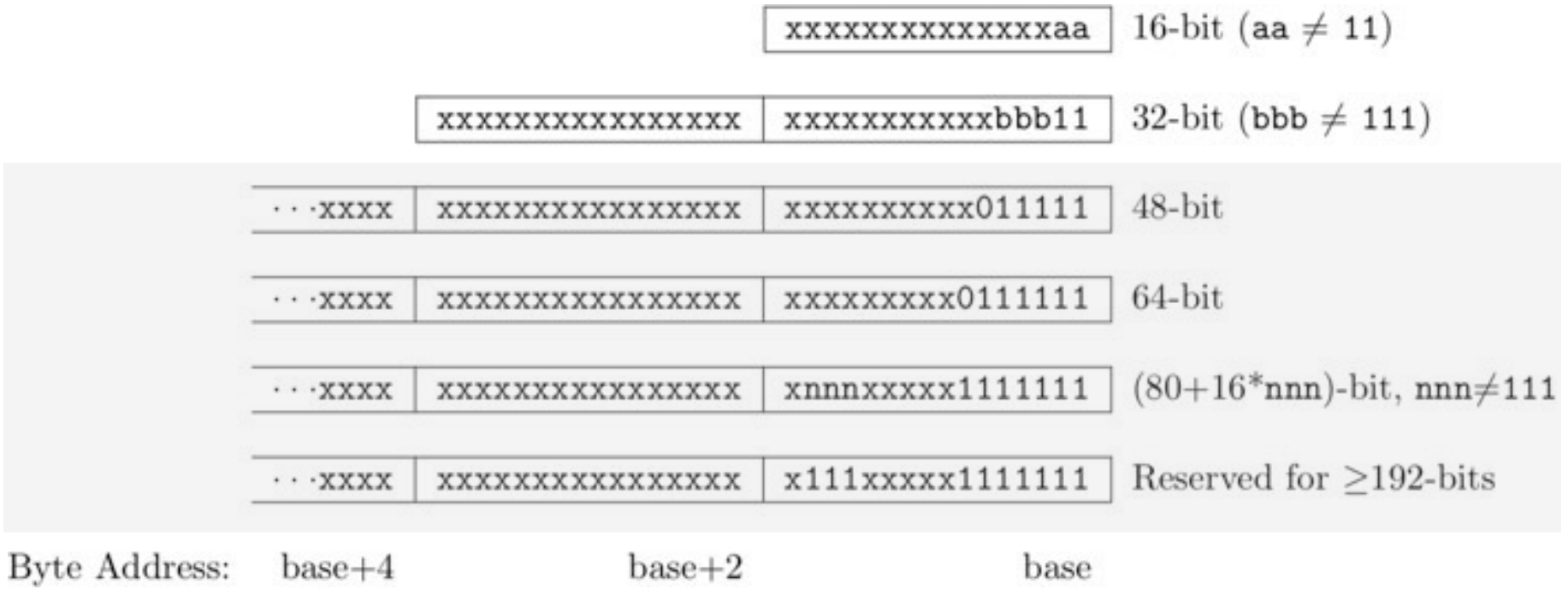
32 registre floating-point (FP) (**f0-f31**)
- fiecare poate conține o valoare single- sau double-precision FP (32-bit sau 64-bit IEEE FP)

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

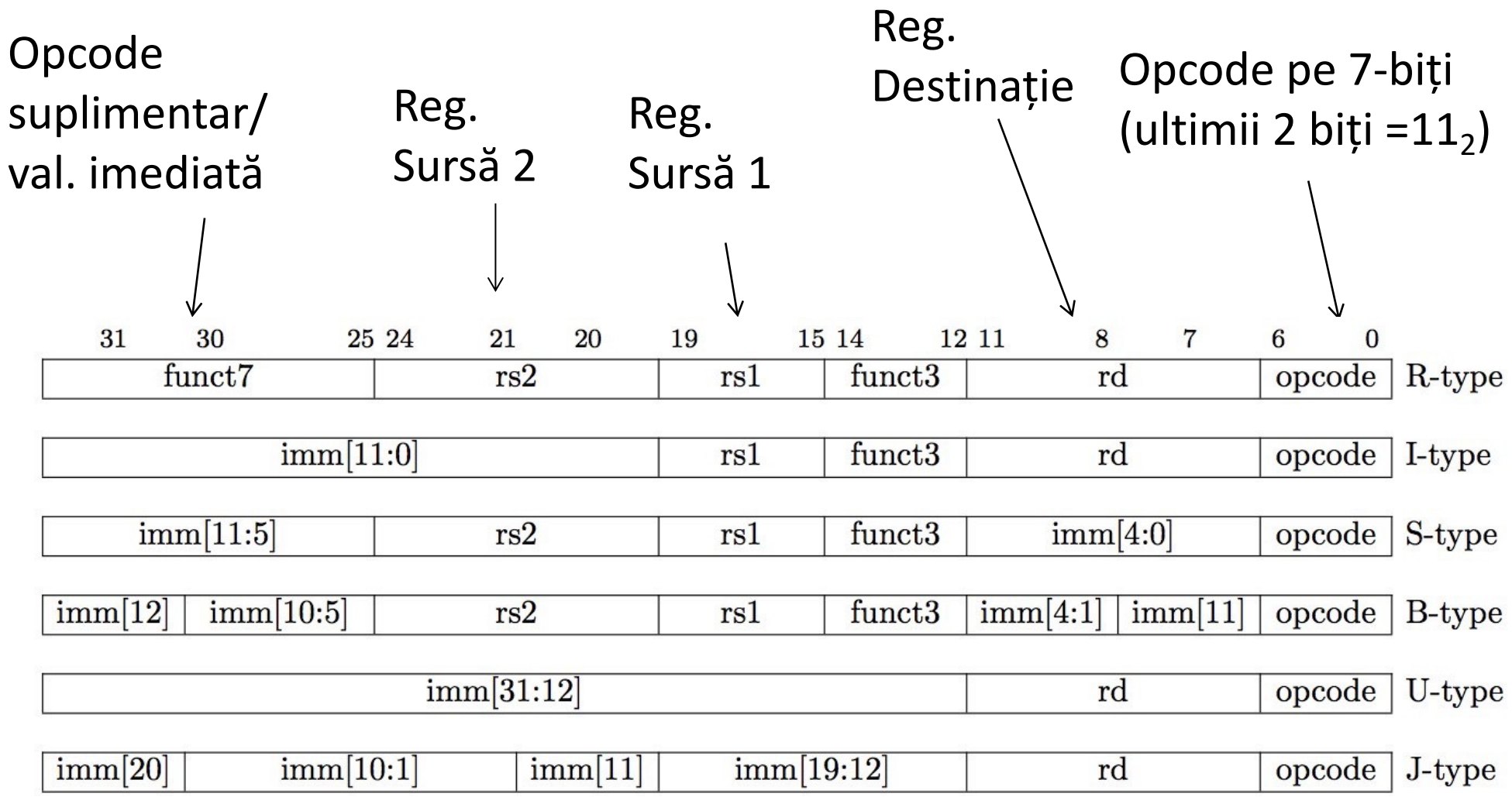


RISC-V Codificarea Instrucțiunilor

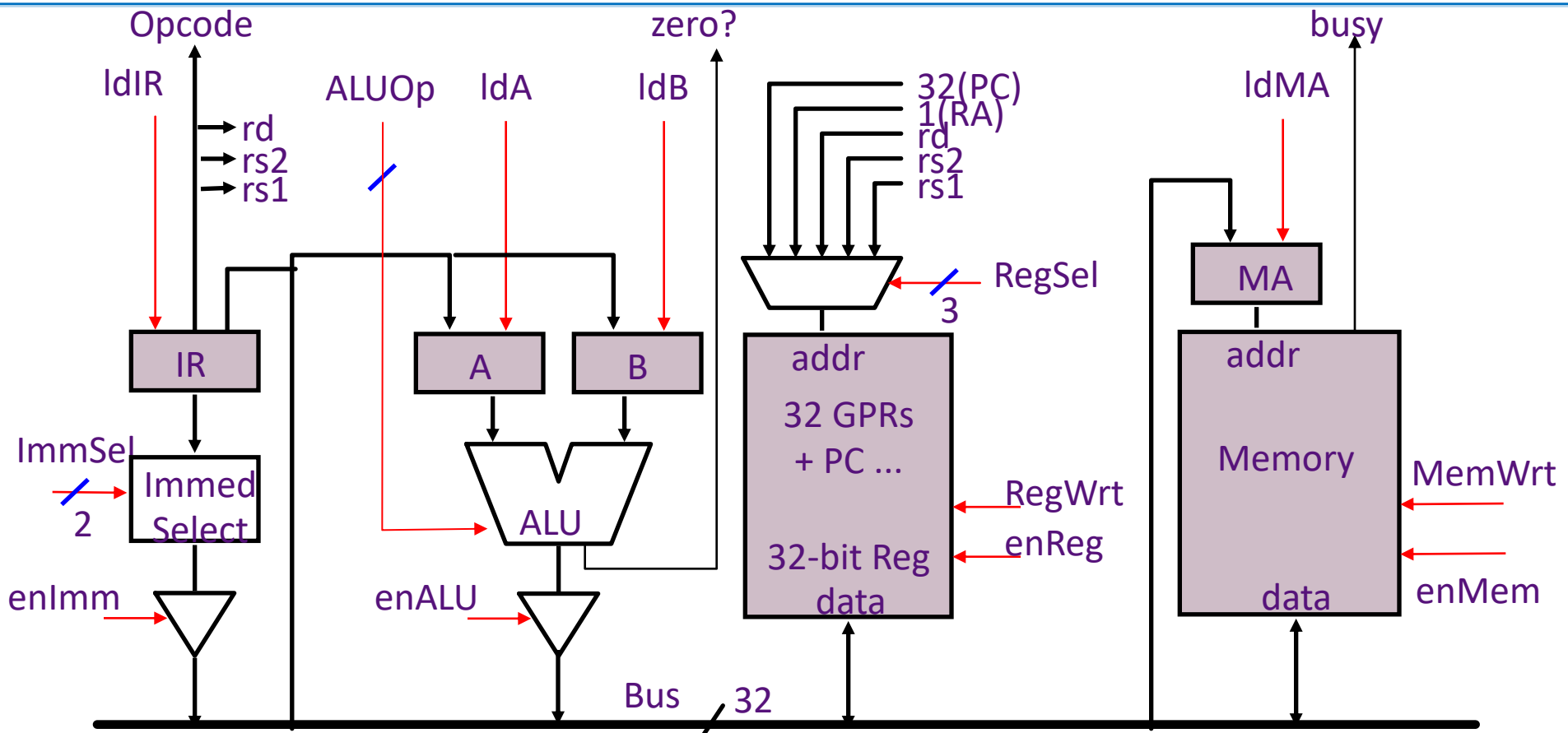
- Suport pentru instrucțiuni de lungime variabilă.
- Setul de bază (RV32) are întotdeauna instrucțiuni de dimensiune fixă, cu ultimii doi biți = 11_2
- Toate instrucțiunile de branch și jump au granularitate de 16 biți (chiar și în ISA de bază, unde toate instrucțiunile au lungimea de 32 de biți)



RISC-V Formatul Instrucțiunilor



RISC-V – descriere generală



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

MA -> PC se traduce prin RegSel = PC; enReg=yes; ldMA= yes

B <- Reg[rs2] *se traduce prin* RegSel = rs2; enReg=yes; ldB = yes

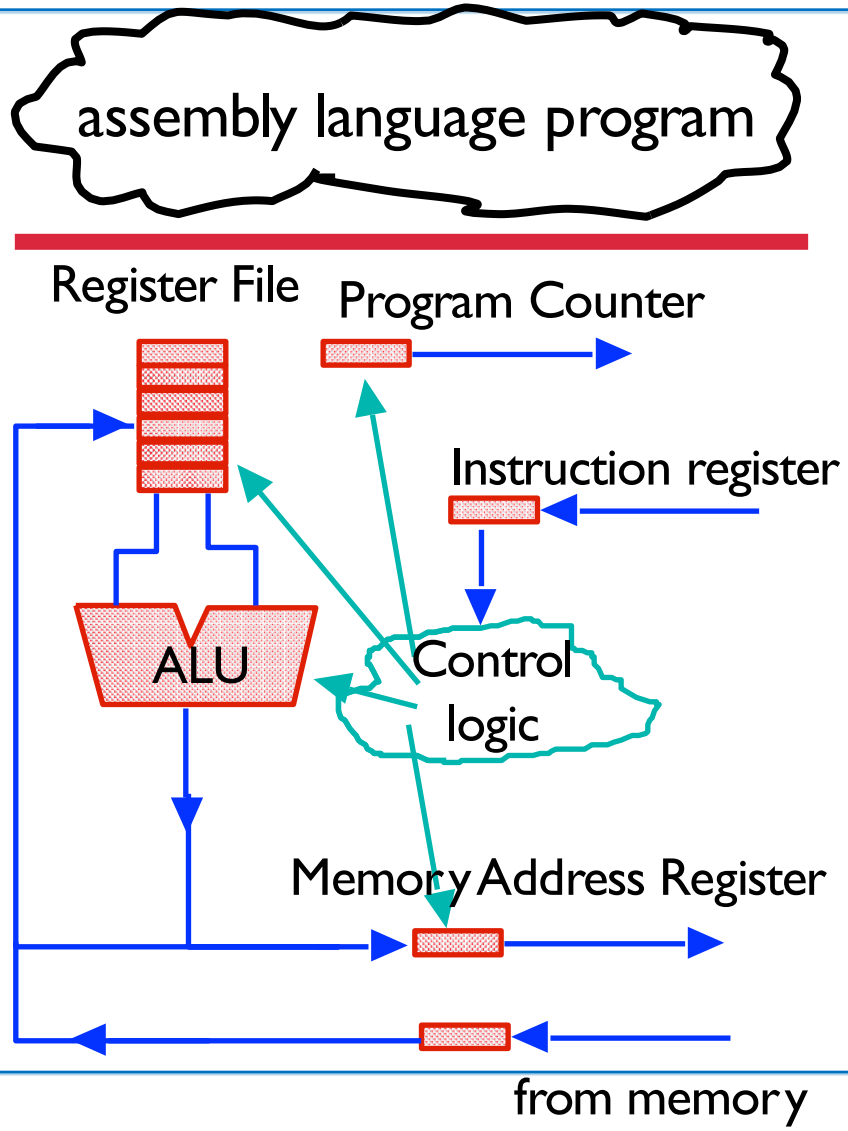
The Big Picture

Limbaj asamblare

Interfața pe care arhitectura o prezintă utilizatorului

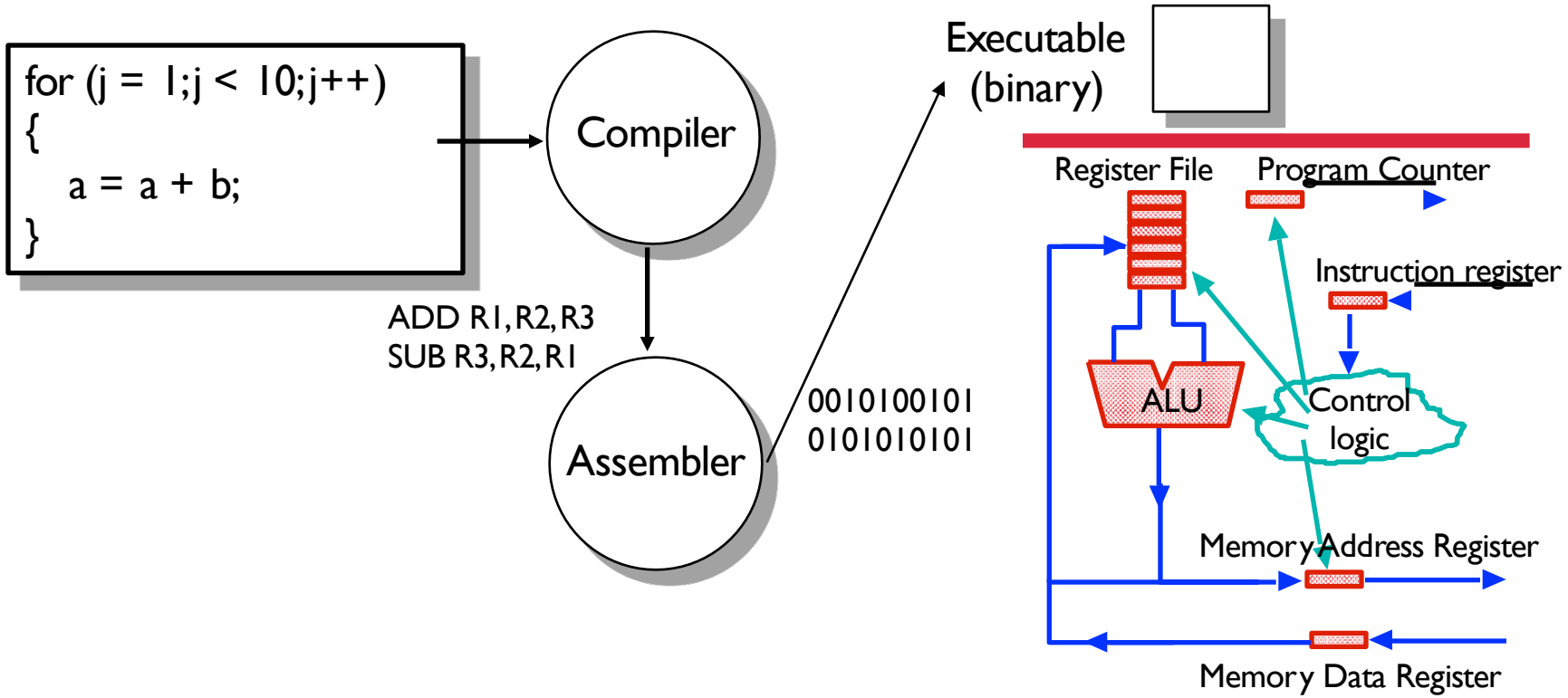
Instrucțiuni ”Low-level” ce folosesc memoria și căile de date pentru a efectua operațiile de bază

- 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 compilator
- 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

- Registrele pot stoca variabile temporare
- Registrele sunt mai rapide decât memoria
- Traficul cu memoria este redus, iar execuția programului este accelerată (pentru că registrele sunt mia rapide ca memoria)
- Densitatea codului este îmbunătățită (pentru că registrele sunt adresate mai ușor decât memoria)

Predomina GPR Machines

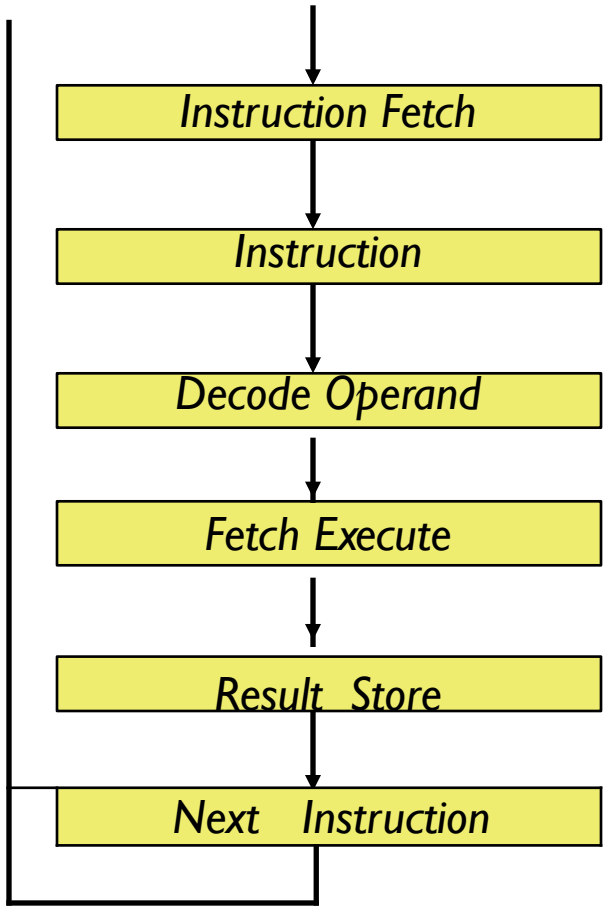
Toate mașinile de calcul folosesc registre generale

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



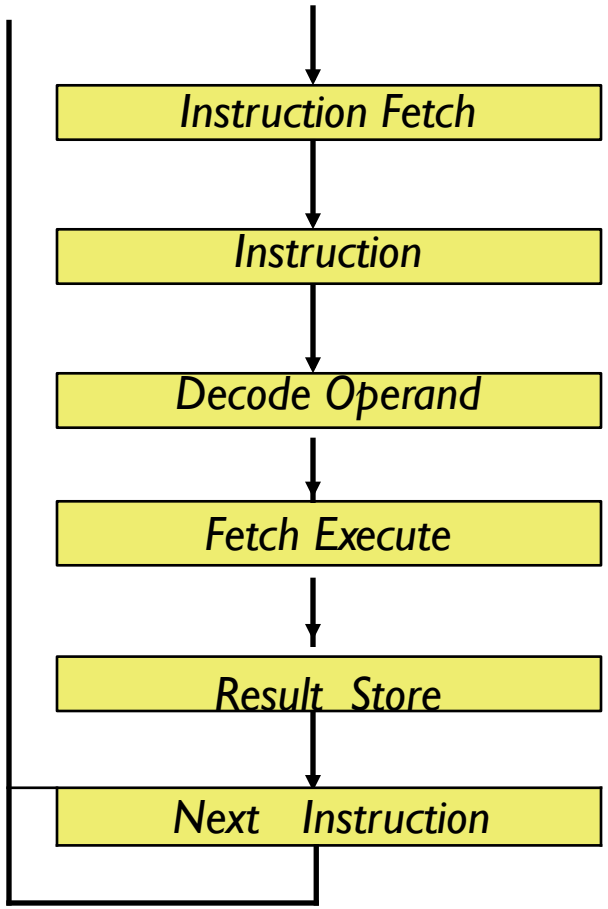
Model secvențial de execuție

- 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

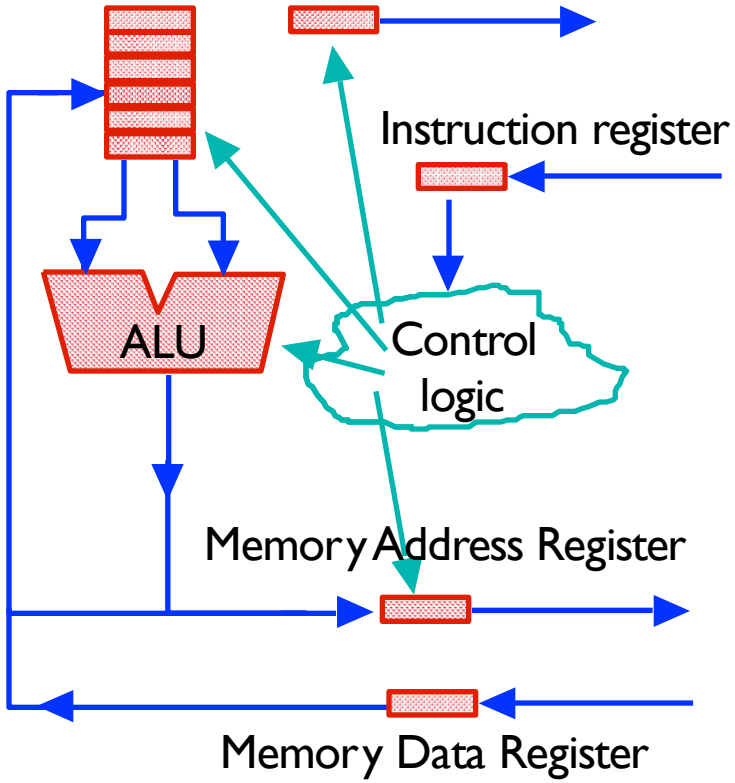
Op-codes and DataTypes

Addressing Modes

Instruction Sequencing

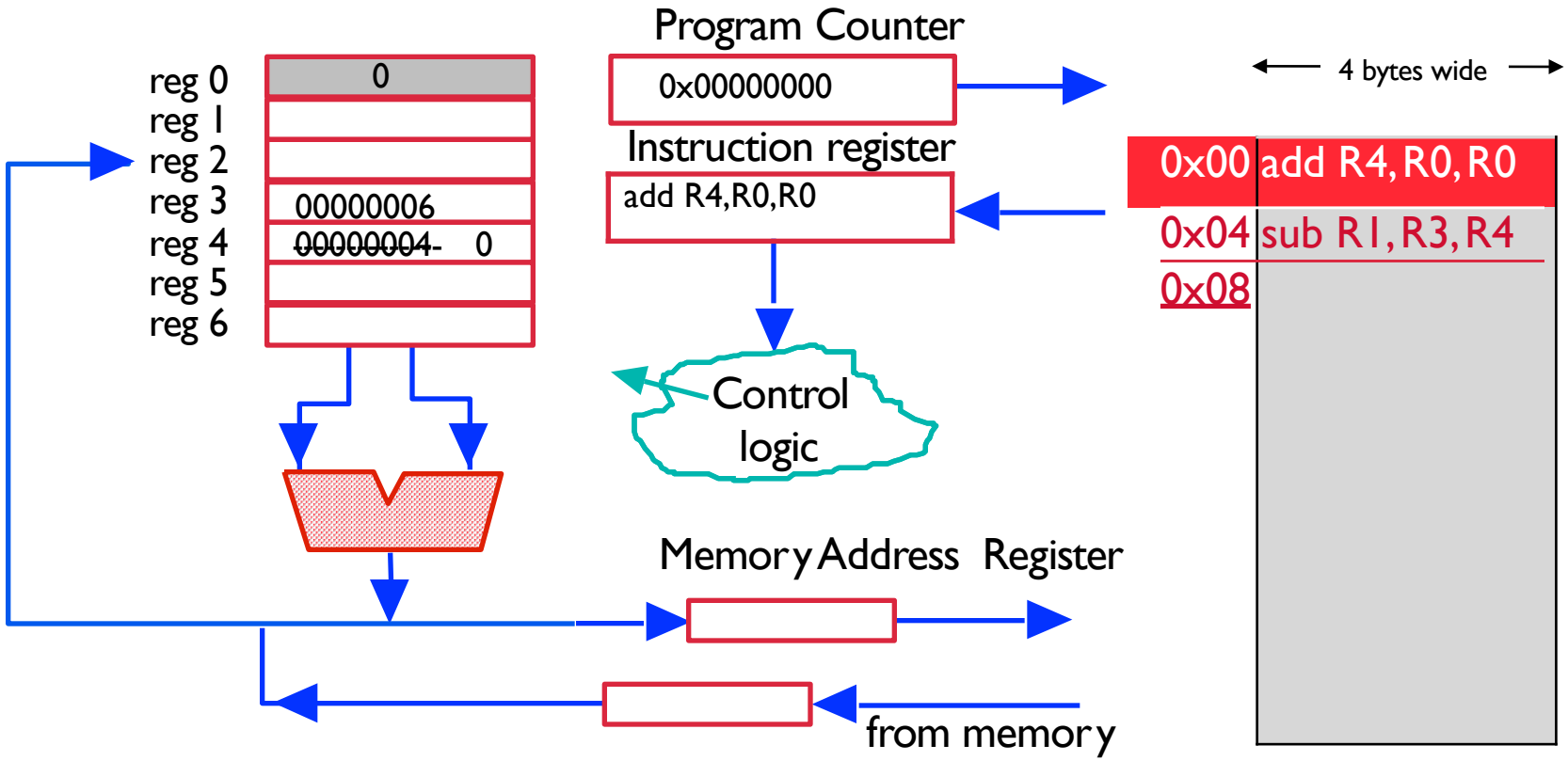
Execuția unei instrucțiuni

- 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

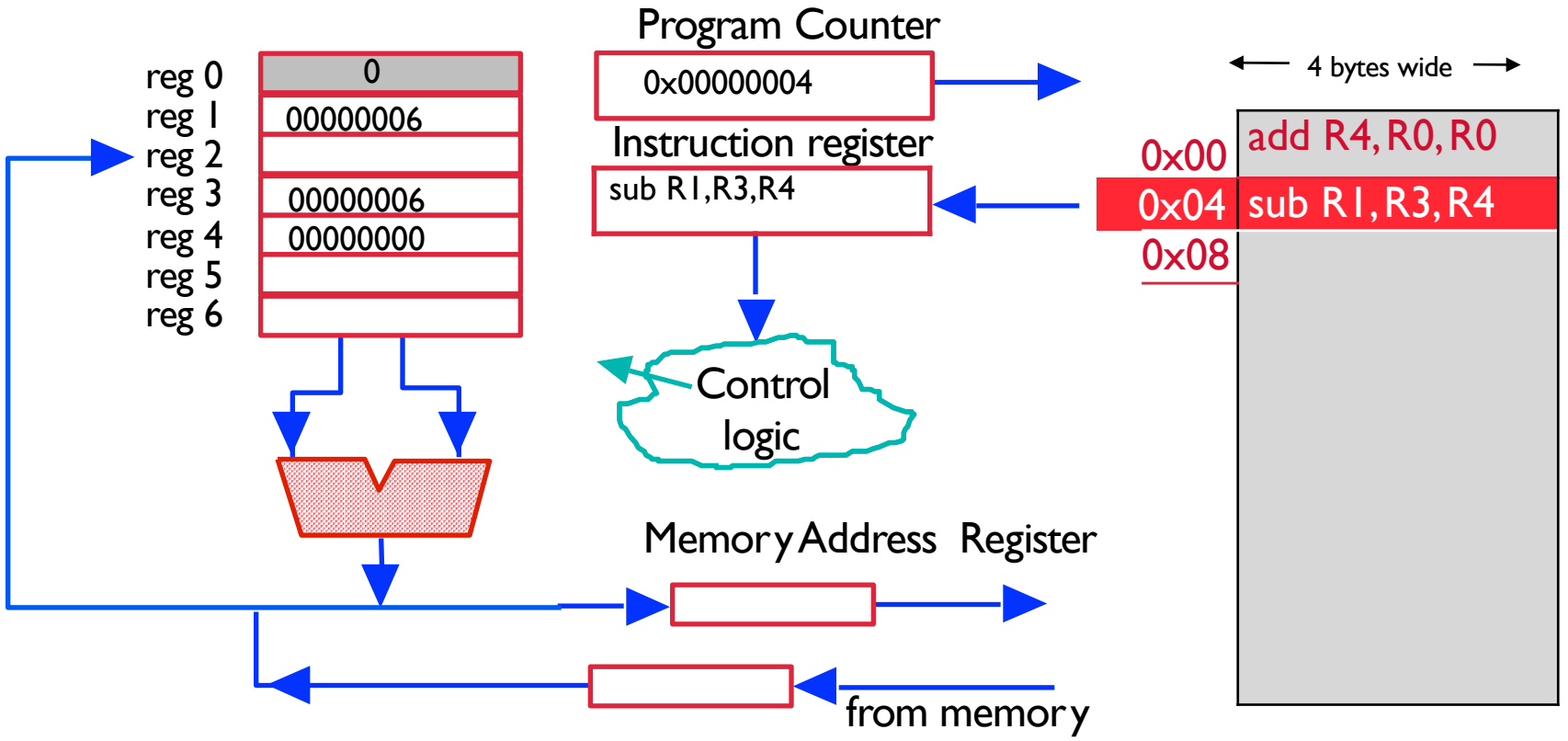


0x00	00101101
0x01	00100001
0x02	00110000
0x03	00001111
0x04	11010101
0x05	01010101
0x06	00101010
0x07	01010101
0x08	11110011
0x09	00111100
0x0A	00001100
0x0B	00000000
0x0C	00011000
0x0D	11111111

Execuție din tabela de registre



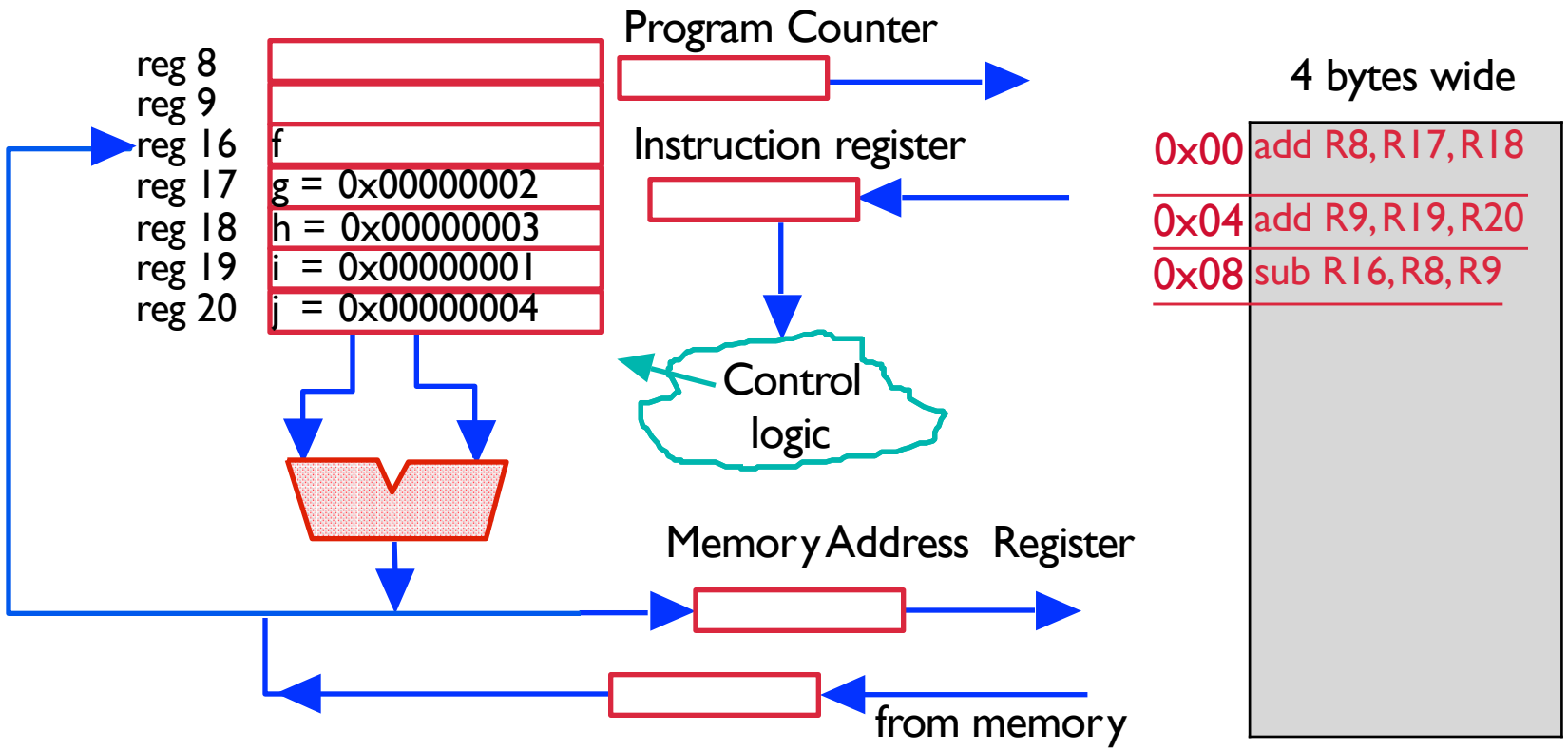
Execuție din tabela de registre



Alt exemplu

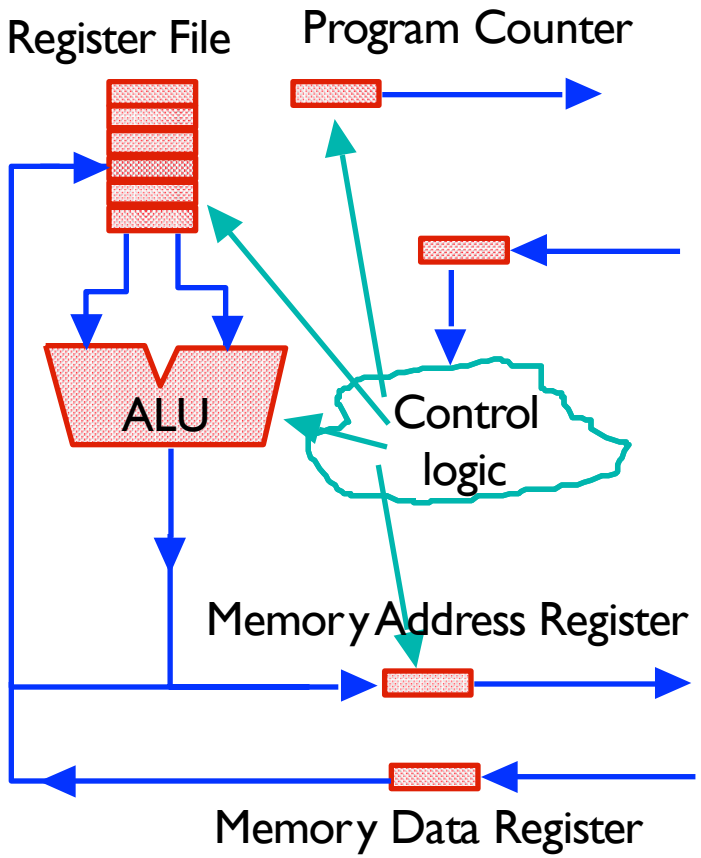
f = (g + h) - (i + j)

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



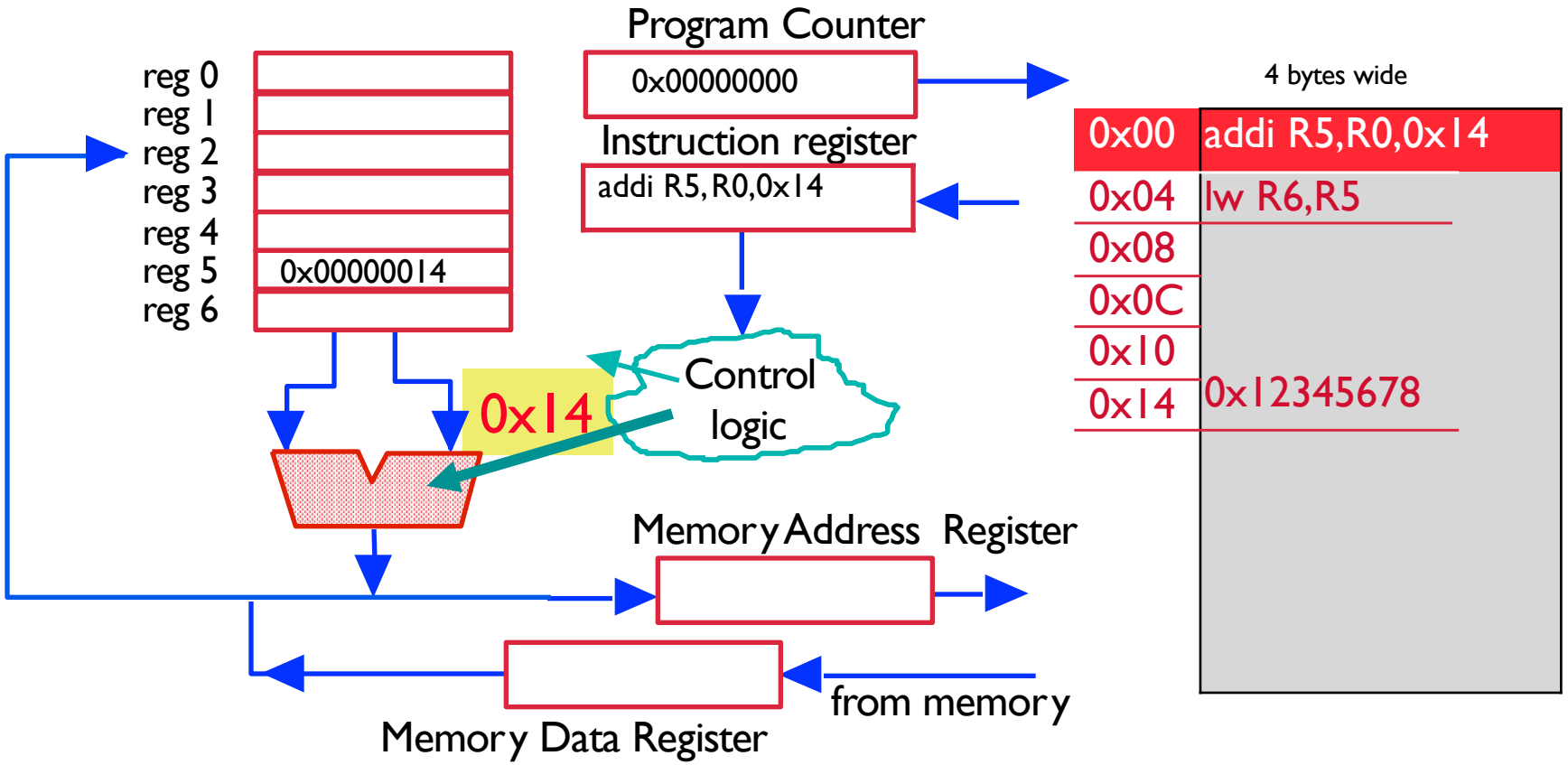
0x00	00101101
0x01	00100001
0x02	00110000
0x03	00001111
0x04	11010101
0x05	01010101
0x06	00101010
0x07	01010101
0x08	11110011
0x09	00111100
0x0A	00001100
0x0B	00000000
0x0C	00011000
0x0D	11111111

Operații cu memoria - Load

R6 ⇐mem[0x14]

Presupunem &A = 0x14

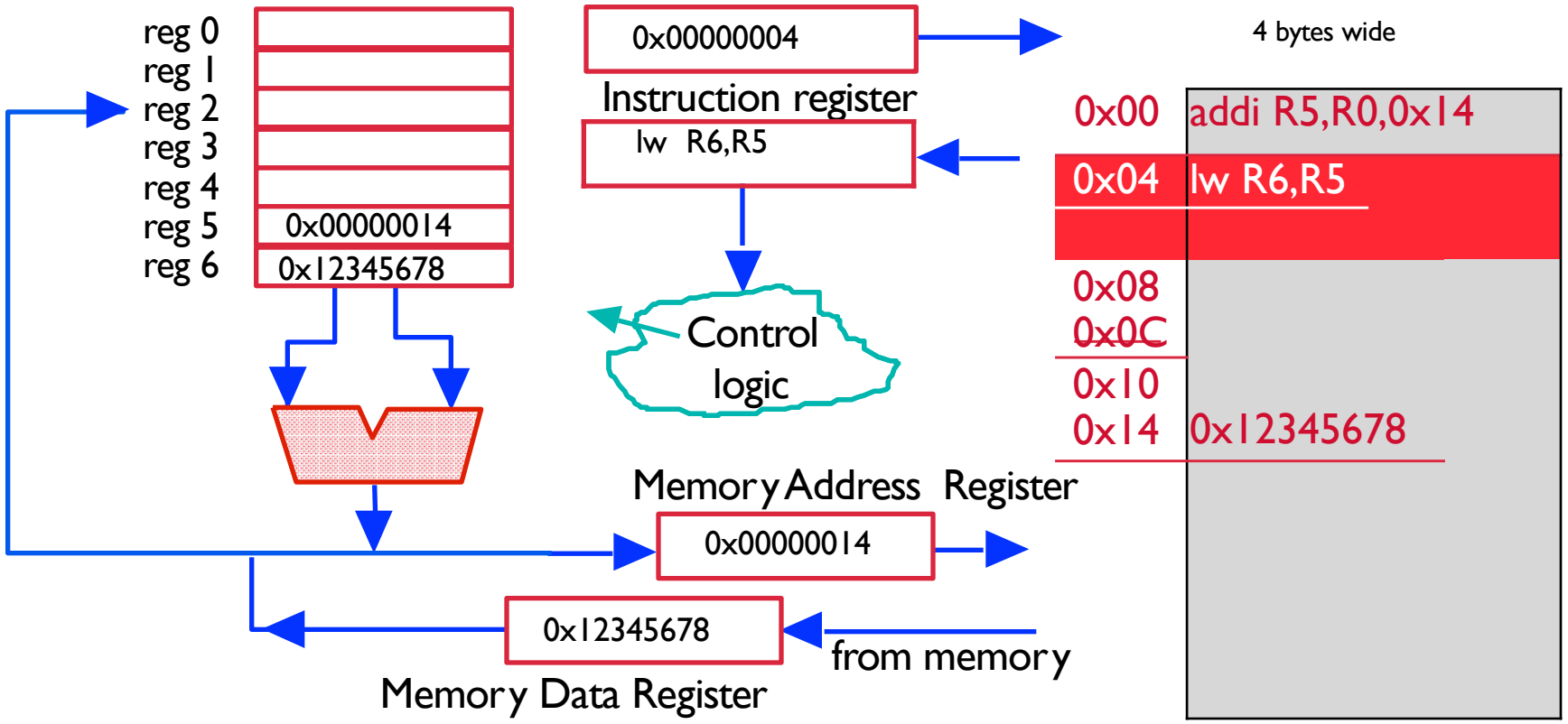
addi: Adună constanta pe 16 biți la registrul sursă



Operații cu memoria - Load

R6 ⇐mem[0x14]

Presupunem &A = 0x14

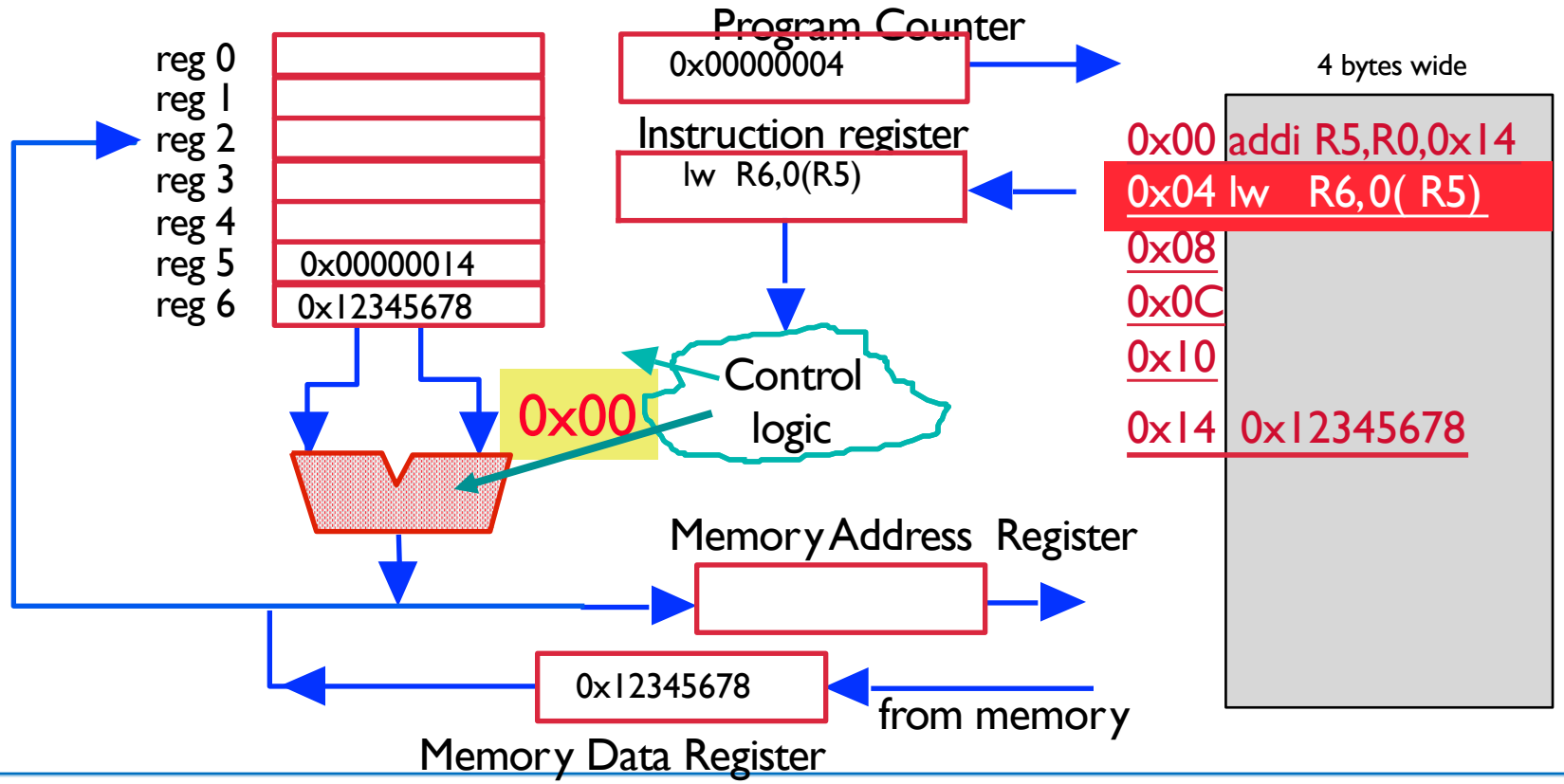


Operații cu memoria - Load

Adresa poate fi calculată și prin adăugarea unui offset registrului

LW R6, 0(R5)

R6 ⇐memory[0 + R5]

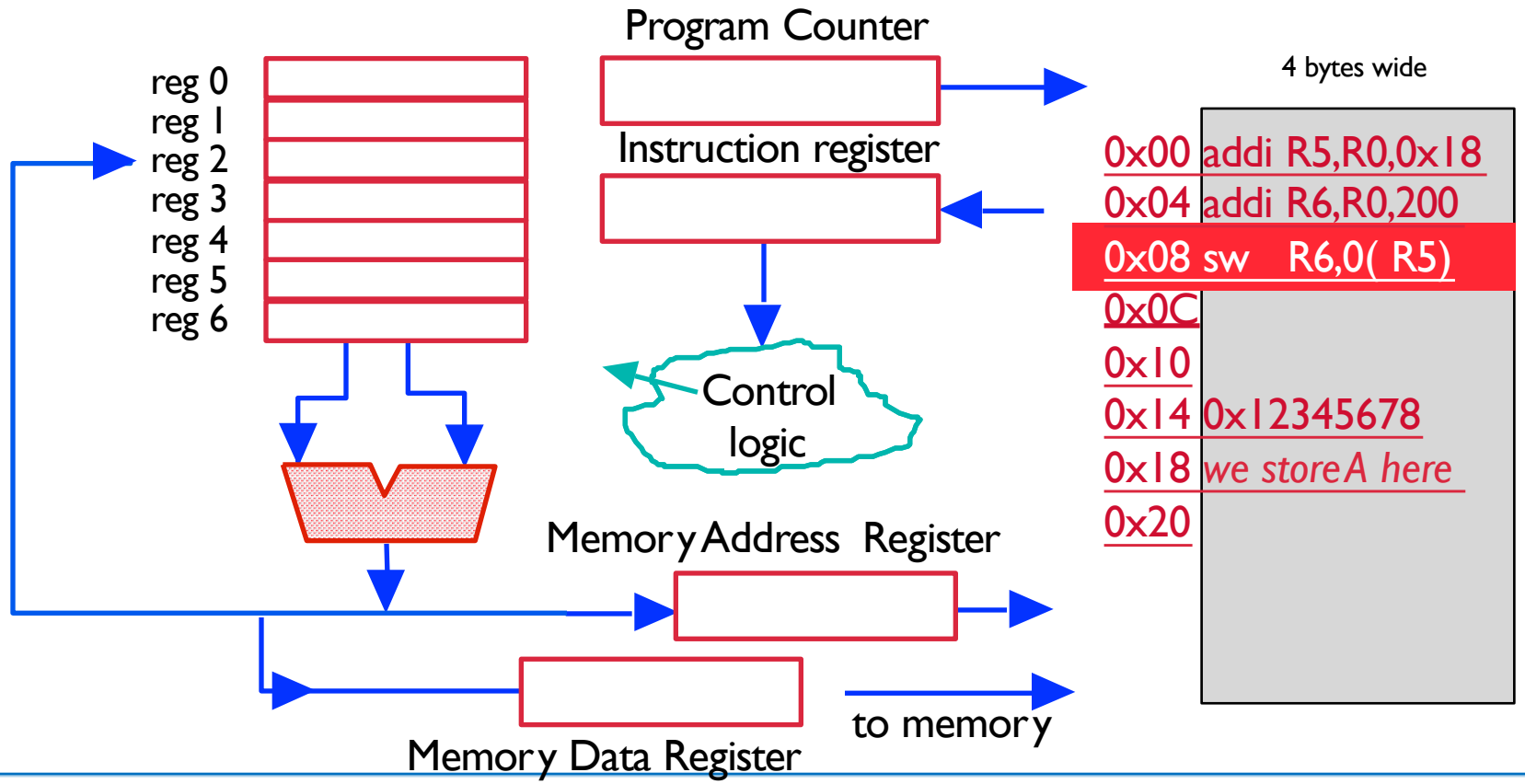


Operații cu memoria - Store

Stocarea datelor în memorie se face la fel

A = 200; presupunem &A = 0x18

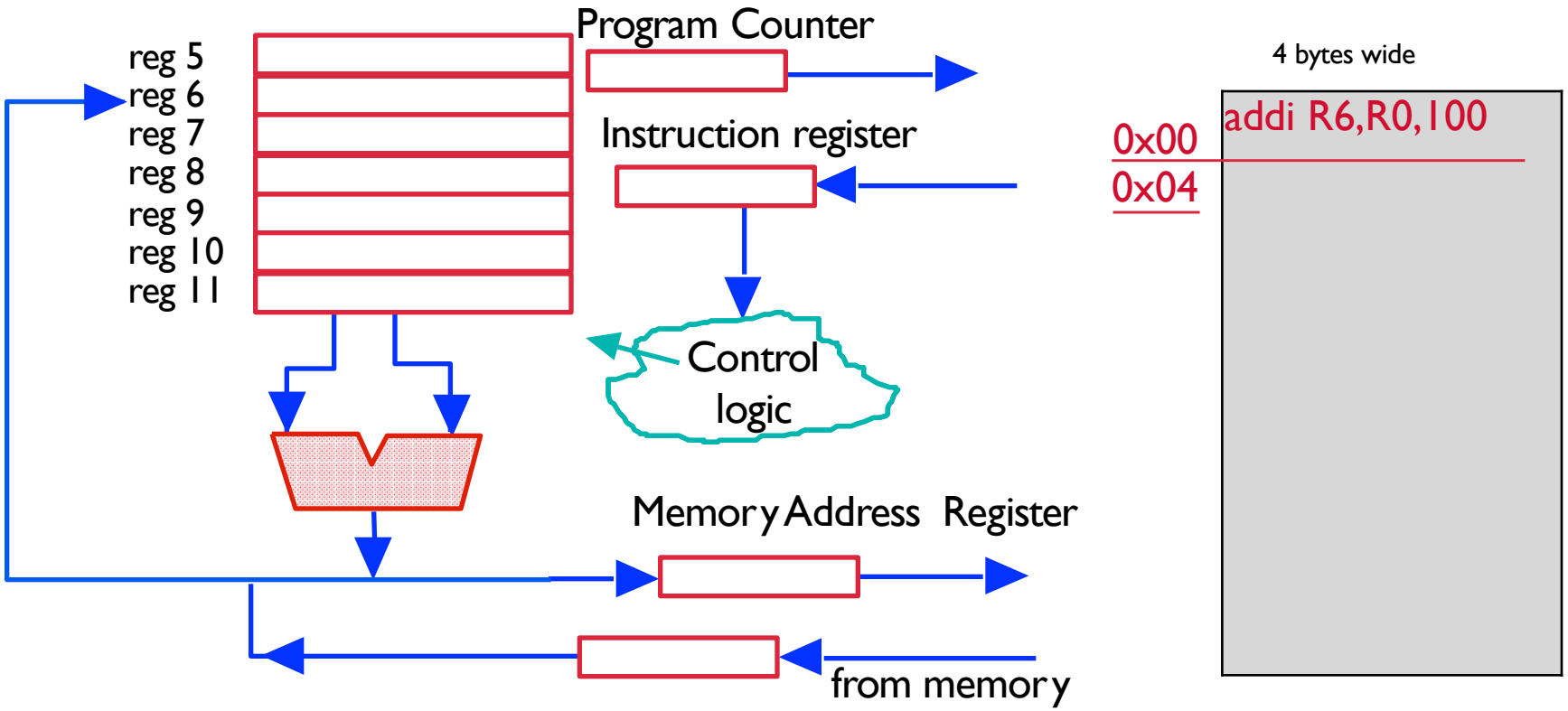
mem[0x18] ⇐200



Încărcarea valorilor imediate

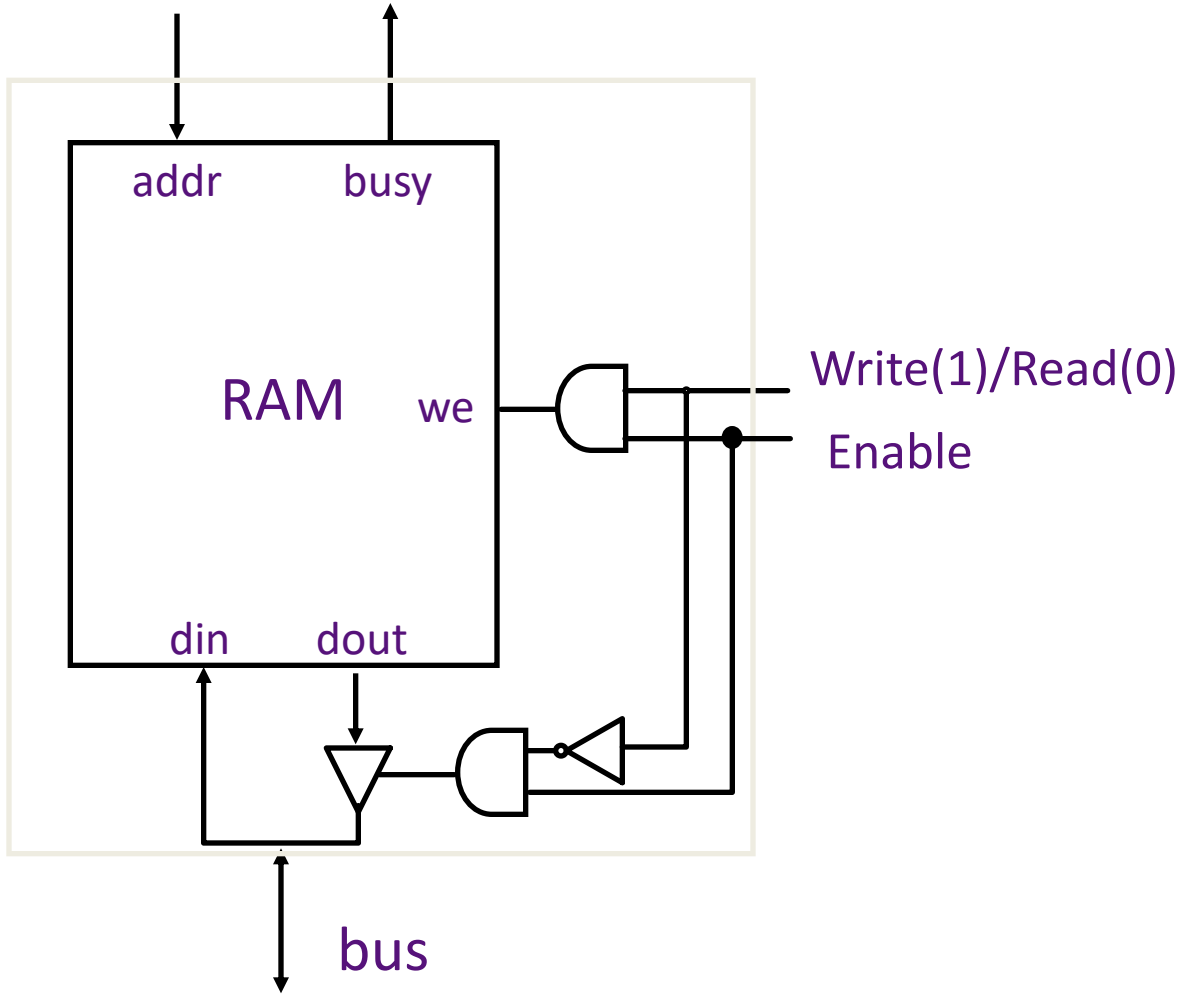
Cum punem o constantă (valoare imediată) în registru?

Scriem valoarea 100 în R6: R6 ⇐ R0 + 100 = 0+100 = 100



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 <- PC
PC <- A + 4
IR <- Memory
dispatch on Opcode

Poate fi văzută ca
O macroinstrucțiune

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

ALUi:
A <- Reg[rs1]
B <- Imm *sign extension*
Reg[rd] <- Opcode(A,B)
do instruction fetch

Fragmente de Microprogram *(cont.)*

LW:
A <- Reg[rs1]
B <-Imm
MA <- A + B
Reg[rd] <- Memory
do instruction fetch

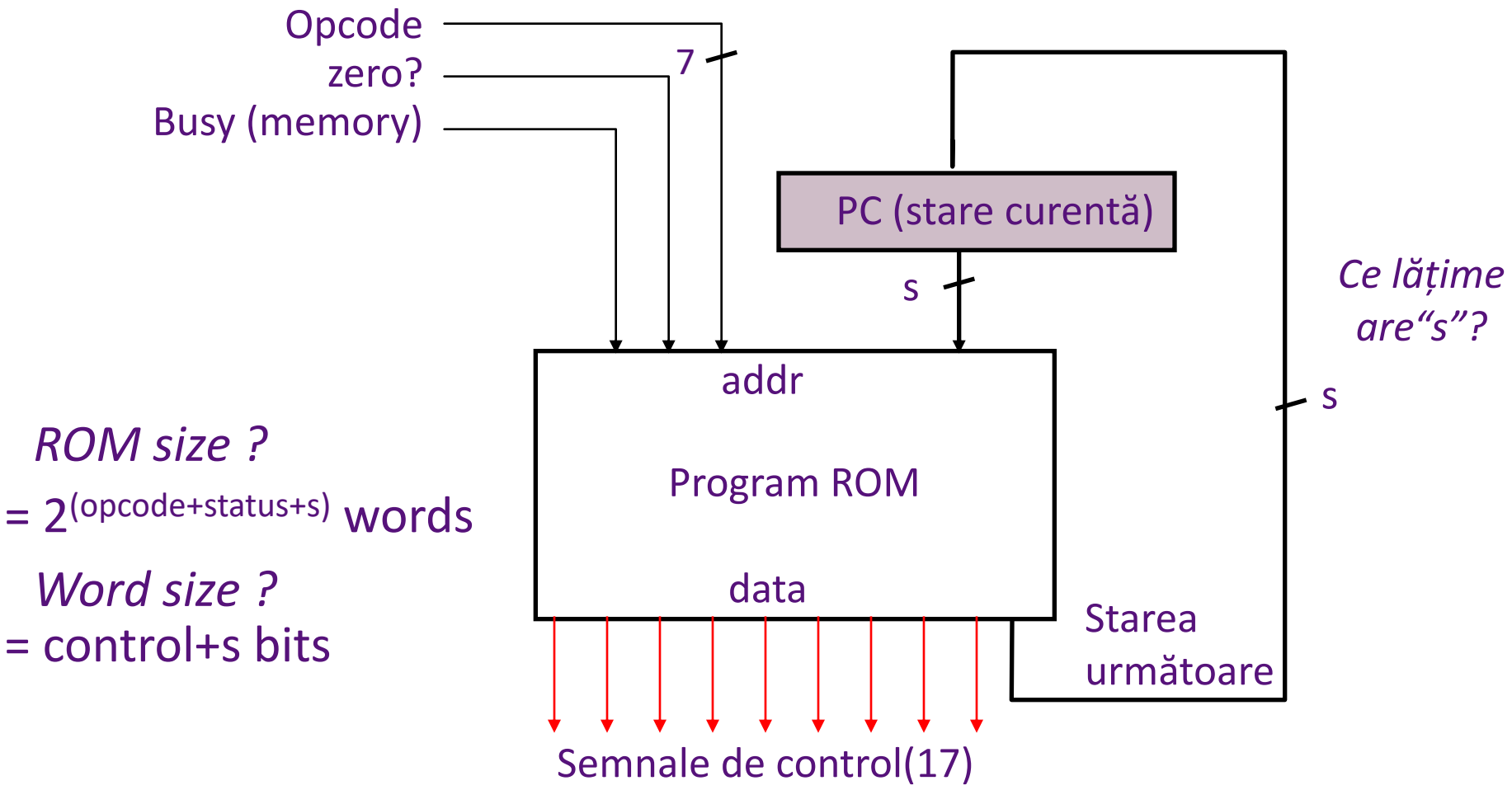
J:
A <- A - 4 Get original PC back in A
B <- IR
PC <- JumpTarg(A,B)
do instruction fetch

JumpTarg(A,B) =
{A + (B[31:7]<<1)}

beq:
A <- Reg[rs1]
B <- Reg[rs2]
If A==B *then go to* bz-taken
do instruction fetch

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
do instruction fetch

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



ROM size ?

= $2^{(opcode+status+s)}$ words

Word size ?

= control+s bits

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

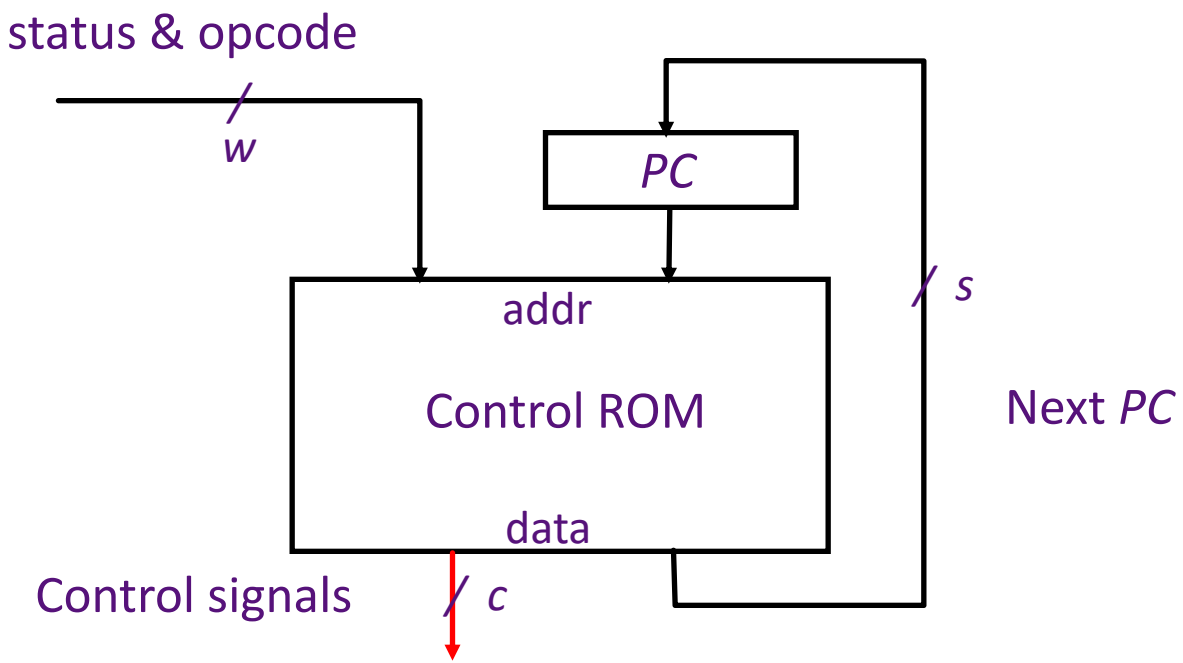
State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA,A <- PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	ALU	*	*	PC <- A + 4	ALU_0
$fetch_2$	ALUi	*	*	PC <- A + 4	$ALUi_0$
$fetch_2$	LW	*	*	PC <- A + 4	LW_0
$fetch_2$	SW	*	*	PC <- A + 4	SW_0
$fetch_2$	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

- Instruction fetch sequence 3 common steps
- ~12 instruction groups
- Each group takes ~5 steps (1 for dispatch)
- Total steps 3+12*5 = 63, needs 6 bits for µPC
- Opcode is 5 bits, ~18 control signals
- Total size = $2^{(6+5+2)}$x(6+18)=2^{13}x24 = ~25KiB!



size = $2^{(w+s)}$ x (c + s)

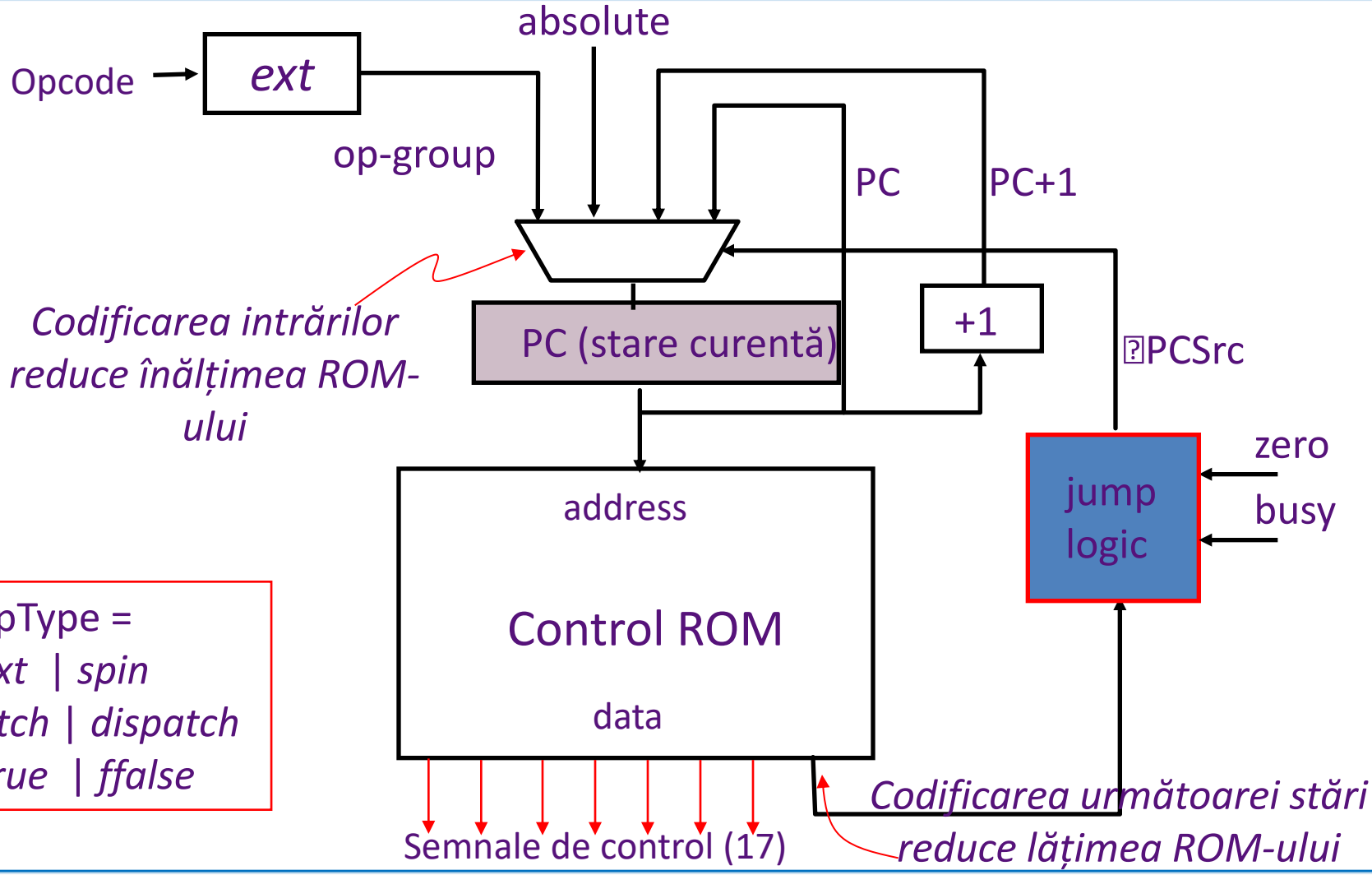
Reducerea dimensiunii memoriei de control

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 <- Reg[rs1]	next
LW_1	B <- Imm	next
LW_2	MA <- A+B	next
LW_3	Reg[rd] <- Memory	spin
LW_4		fetch
SW_0	A <- Reg[rs1]	next
SW_1	B <- BImm	next
SW_2	MA <- A+B	next
SW_3	Memory <- Reg[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

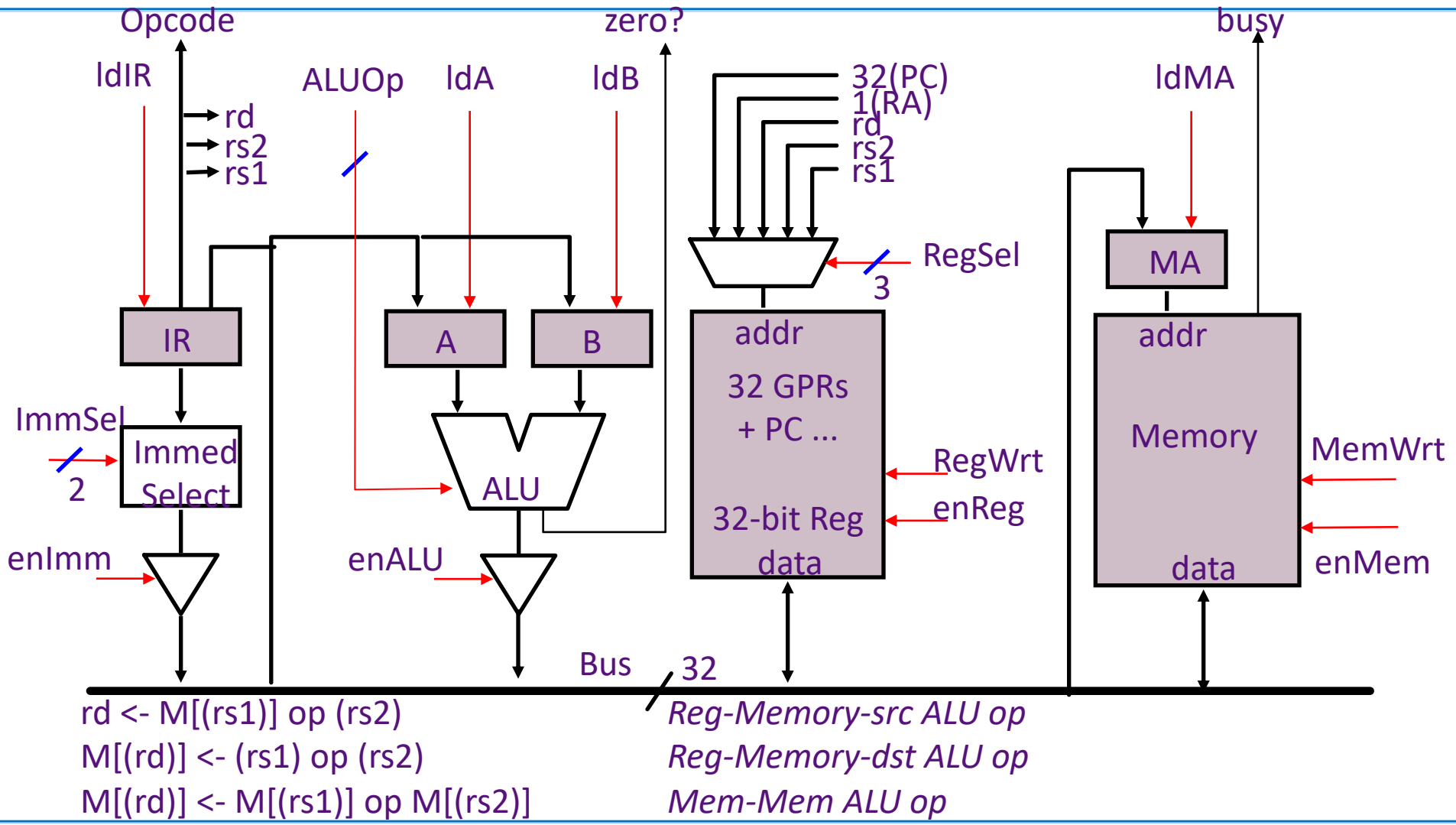
Jumps: *RISC-V-Controller-2*

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

VAX 11-780 Microcode

```
; P1WFUD.I  [600,1205]      MICRO2  1F(12)    26-May-81  14:58:1      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
6557K    0  U 11F4, 0811,2035,0180,F910,0000,0CD8 ;29748        CALL,J/MPUSH                    ;PUSH REGISTERS
                                                ;29749
                                                ;29750          ;-------------------------------;RETURN FROM MPUSH
                                                ;29751          CACHE_D[LONG],                  ;PUSH PC
6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A ;29752        LAB_R[SP]                       ; BY SP
                                                ;29753
                                                ;29754          ;-------------------------------;
6856K    0  U 134A, 0018,0000,0180,FAF0,0200,134C ;29755  CALL.8: R[SP]&VA_LA-K[,8]              ;UPDATE SP FOR PUSH OF PC &
                                                ;29756
                                                ;29757          ;-------------------------------;
6856K    0  U 134C, 0800,003C,0180,FA68,0000,11F8 ;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
6856K   21M U 11F8, 0000,003D,6D80,3270,0084,6CD9 ;29764        CALL,J/PSHSP
                                                ;29765
                                                ;29766          ;-------------------------------;
                                                ;29767          D_R[AP],                        ;READY TO PUSH AP
6856K    0  U 11F9, 0800,003C,3DF0,2E60,0000,134D ;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>
6856K   21M U 134D, 0019,2024,8DC0,3270,0000,134E ;29773        LAB_R[SP]                       ;GET SP INTO LATCHES AGAIN
                                                ;29774
                                                ;29775          ;-------------------------------;
6856K    0  U 134E, 2010,0038,0180,F909,4200,1350 ;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
6856K    0  U 1350, 0D10,0038,0DC0,6114,0084,9351 ;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_RC[T4],                       ;GET LOW BITS OF OLD SP TO Q<1:0>
6856K    0  U 1351, 0D10,0038,F5C0,F920,0084,9352 ;29787        SC_SC+K[,A]                     ;PUT -3 IN SC
                                                ;29788
```


Opcode
zero?
busy
ldIR
ALUOp
ldA
ldB
32(PC)
1(RA)
rd
rs2
rs1
ldMA
rd
rs2
rs1
RegSel
3
MA
IR
A
B
addr
32 GPRs
+ PC ...
32-bit Reg
data
addr
Memory
data
ImmSel
2
Immed
Select
ALU
RegWrt
enReg
MemWrt
enMem
enImm
enALU
Bus
32
rd <- M[(rs1)] op (rs2)
M[(rd)] <- (rs1) op (rs2)
M[(rd)] <- M[(rs1)] 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)] <- M[(rs1)] op M[(rs2)]	
$ALUMM_0$	MA <- Reg[rs1]	next
$ALUMM_1$	A <- Memory	spin
$ALUMM_2$	MA <- Reg[rs2]	next
$ALUMM_3$	B <- Memory	spin
$ALUMM_4$	MA <- Reg[rd]	next
$ALUMM_5$	Memory <- 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

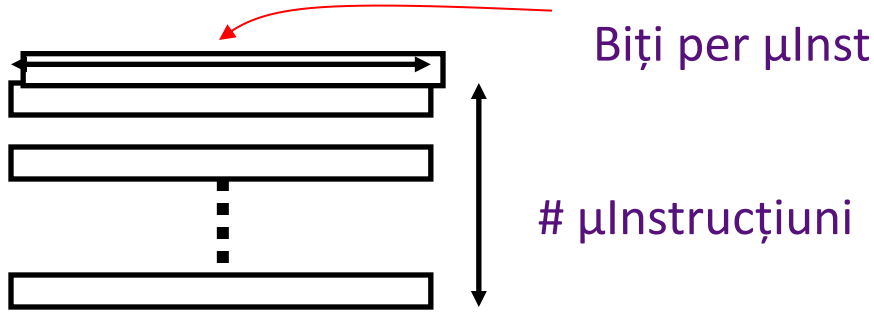
Durata ciclului ?

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

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

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

µCod orizontal vs. vertical

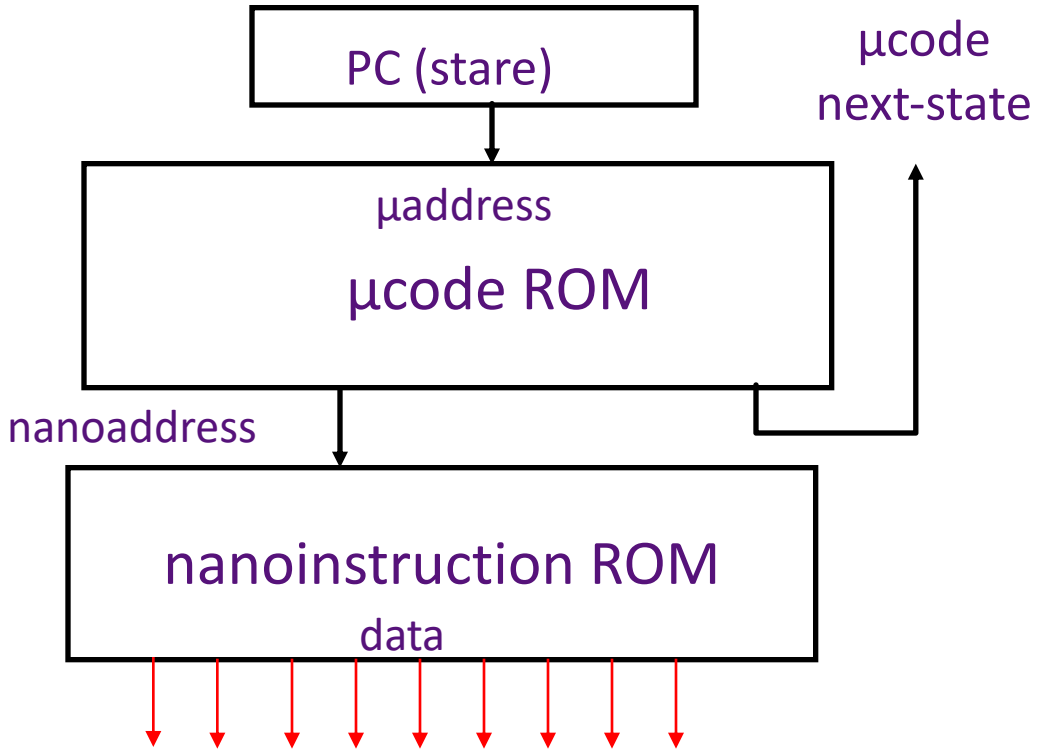


- µcodul orizontalizat are µinstrucțiuni mai lungi
  - Mai multe operații per µinstrucțiune
  - Mai puțini pași în microcod pe macroinstrucțiune
  - Sparse encoding ⇒ mai mulți biți
- µcodul verticalizat are µinstrucțiuni scurte
  - De obicei, o singură operație de transfer date per µinstrucțiune
    - 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.,

ALU_0 A <- Reg[rs1]
...
$ALUi_0$ 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 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 mies 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 Zen, Intel Sky Lake, 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)
  - Intel a trebuit să (re)găsească ingineri pentru a rescrie microcodul procesoarelor sale pentru a face patch la vulnerabilitățile expuse de Spectre/Meltdown

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