Arhitectura Sistemelor de Calcul

Modificat: 22-Oct-23

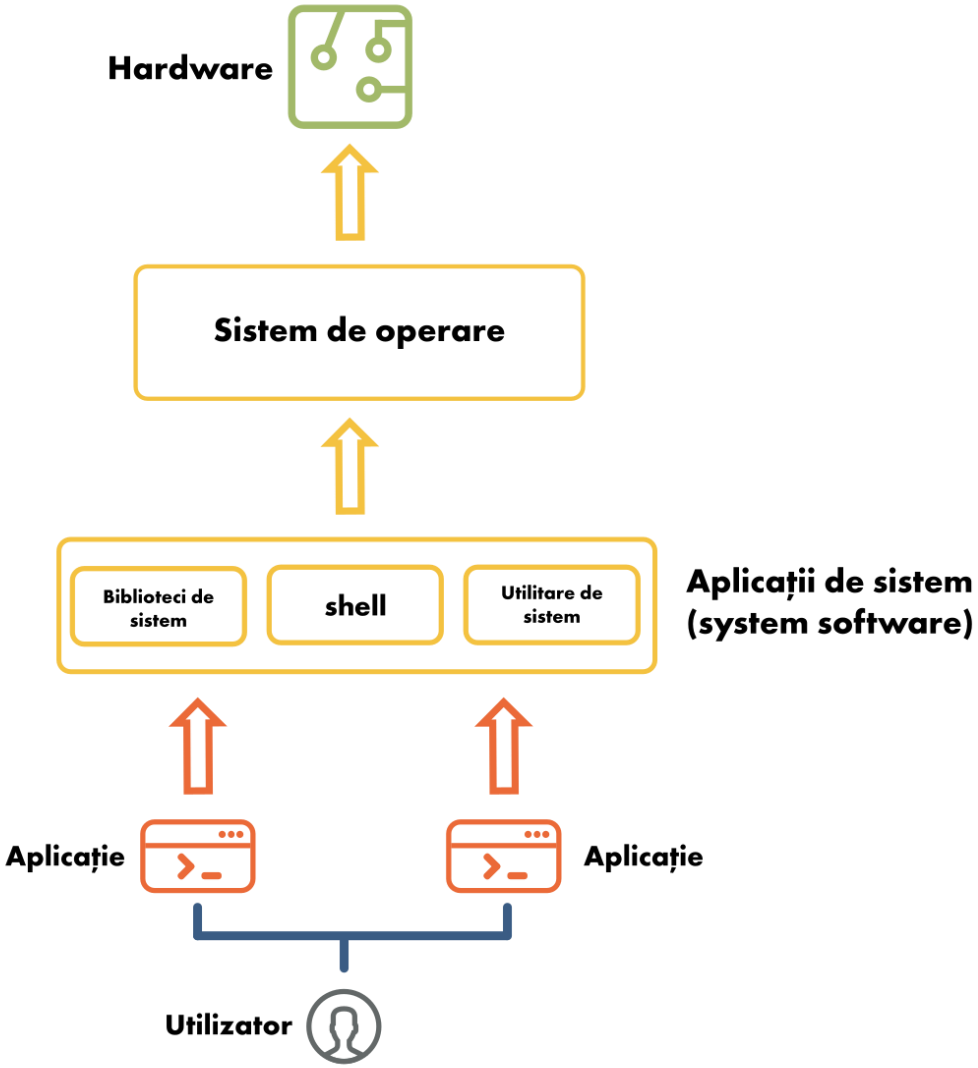
Cuprins curs 1

- Imagine de ansamblu asupra sistemului
- Ce este limbajul de asamblare
  - Limbajul mașină
- Avantajele limbajelor de nivel înalt
  - Viteza de dezvoltare
  - Mentenanța ușoară
  - Portabilitate
- De ce assembler?
  - Eficiența în spațiu
  - viteza
  - Acces la hardware
- De ce să știm assembler?
- Arhitectura Sistemelor de Calcul = curs anul 3
- Hello World

De citit:
Capitolul 1
Capitolul 2: 2.1, 2.2 2.5 2.6
Anexa B

SOFTWARE ȘI HARDWARE

Reminder: Utilizator – software - hardware



Cum se vede sistemul la utilizator

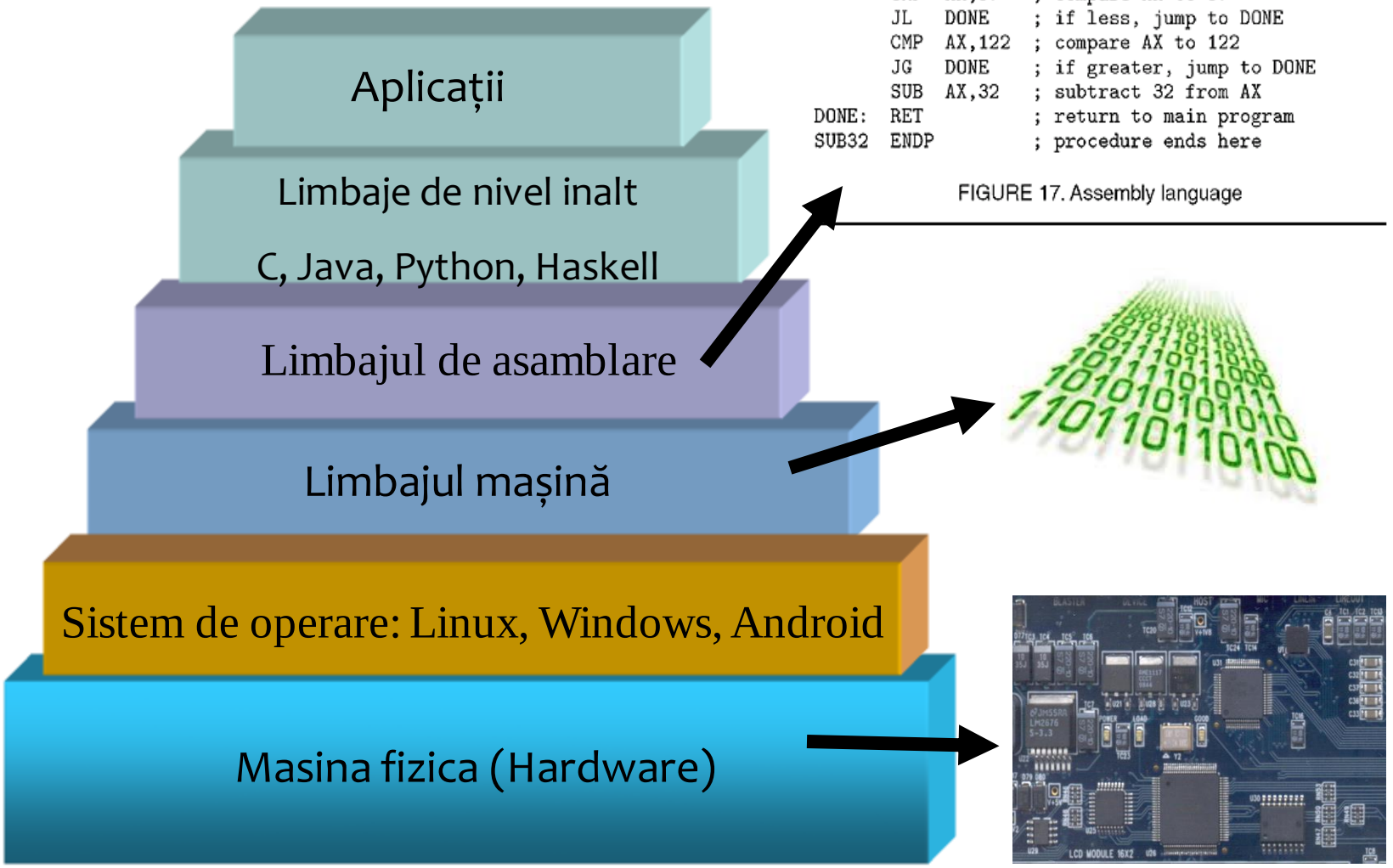
- Depinde de gradul de abstractizare software
- Ierarhie de 6 niveluri
  - Vârful ierarhiei izolează utilizatorul de hardware
  - *Independente de sistem:* primele două
  - *Dependente de sistem:* ultimele trei
    - » Limbajele asamblare/mașină sunt specifice procesorului
    - » Corespondență directă între limbajele asamblare/mașină

De la aplicații la hardware

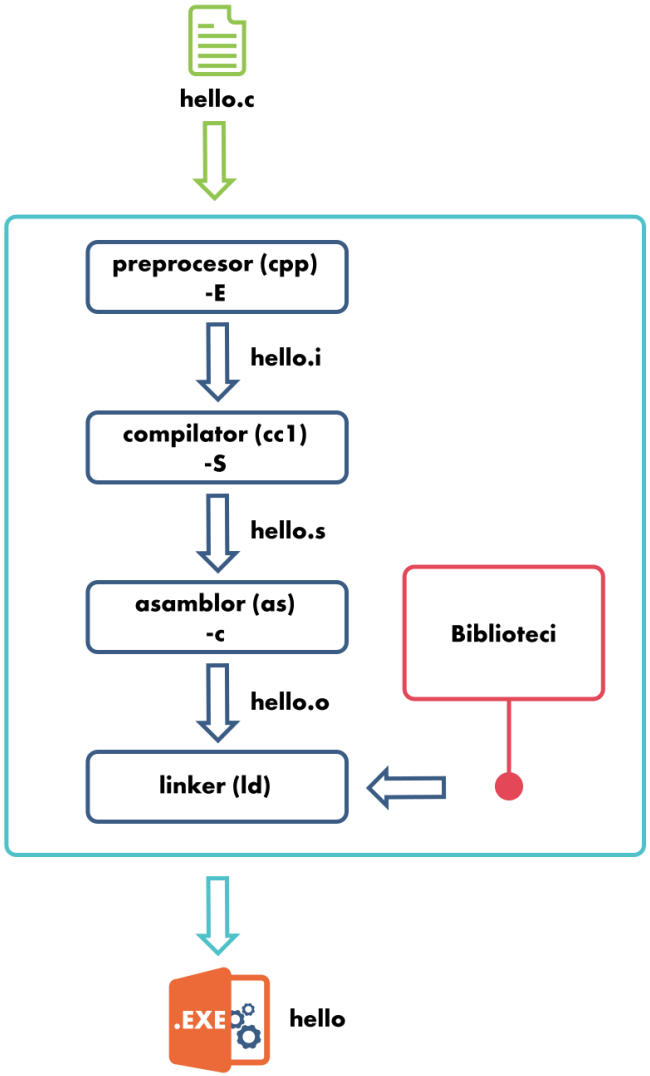
```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32   PROC          ; procedure begins here
        CMP   AX,97   ; compare AX to 97
        JL    DONE    ; if less, jump to DONE
        CMP   AX,122  ; compare AX to 122
        JG    DONE    ; if greater, jump to DONE
        SUB   AX,32   ; subtract 32 from AX
DONE:   RET           ; return to main program
SUB32   ENDP          ; procedure ends here
```

FIGURE 17. Assembly language



Reminder: De la cod sursă la executabil



Cod mașină

- machine code
- "limbajul" procesorului
- Instrucțiuni cod mașină
  - interpretate de procesor
- Executabilele conțin cod mașină
- Codul mașina este generat de compilator
- Limbaj de nivel înalt -> compilator -> cod mașină

LIMBAJUL DE ASAMBLARE

Ce este limbajul de asamblare?

- Limbaj de nivel jos
  - » Fiecare instrucțiune rezolvă un task simplu
- Corespondență directă între limbajele de asamblare/mașină
  - » Pentru majoritatea instrucțiunilor asamblare există un cod mașină echivalent
  - » ***Asamblorul*** traduce codul asamblare în cod mașină
- Influențat direct de setul de instrucțiuni și arhitectura procesorului (CPU)

Instrucțiuni în limbaj de asamblare

- Exemple de instrucțiuni asamblor:

```
inc     result
mov     class_size,45
and     mask1,128
add     marks,10
```

Exemplu MIPS

```
andi   $t2,$t1,15
addu   $t3,$t1,$t2
move   $t2,$t1
```

- Observații
  - » Instrucțiunile asamblor sunt **criptice**
  - » Mnemonice sunt folosite pentru operații

    inc pentru increment, **mov** for move (copiere)
  - » Instrucțiunile asamblor sunt **de nivel jos**
  - » Nu există instrucțiuni de tipul:

```
mov     marks, value
```

C vs limbaj de asambalre

- Unele instrucțiuni de nivel înalt pot fi exprimate printr-o singură instrucțiune asamblor:

Asamblare	C
`inc result`	`result++;`
`mov class_size,45`	`class_size = 45;`
`and mask1,128`	`mask1 &= 128;`
`add marks,10`	`marks += 10;`

C vs limbaj de asamblare (2)

- Majoritatea instrucțiunilor de nivel înalt necesită mai multe instrucțiuni asamblor:

C	Asamblare
`size = value;`	`mov EAX,value` `mov size,EAX`

`sum += x + y + z;`	`mov EAX,sum`
	`add EAX,x`
	`add EAX,y`
	`add EAX,z`
	`mov sum,EAX`

Limbaj de asamblare vs cod mașină (x86)

- Limbajul de asamblare se citește mai ușor decât limbajul mașină (binar)

Asamblare		**Cod mașină**(Hex)
`inc`	`result`	`FF060A00`
`mov`	`class_size,45`	`C7060C002D00`
`and`	`mask,128`	`80260E0080`
`add`	`marks,10`	`83060F000A`

Limbaj de asamblare vs cod mașină (MIPS)

- Exemplu MIPS

Asamblare	Cod mașină (HEX)
`nop`	`00000000`
`move $t2,$t15`	`000A2021`
`andi $t2,$t1,15`	`312A000F`
`addu $t3,$t1,$t2`	`012A5821`

Arhitecturi de procesor

- x86, ARM, MIPS, PowerPC
- Interfața expusă de procesor către software
- În general numite ISA
  - Instruction Set Architecture
- Specificația codului mașină
- Un compilator poate genera fișiere cod mașină pentru diferite ISA de la același cod sursă
- Limbajul de asamblare diferă pentru fiecare arhitectură
  - Este corespondent fiecărui cod mașină (ISA)

Avantajele limbajelor de nivel înalt

- Dezvoltarea este **mai rapidă**
  - » Instrucțiuni de nivel înalt
  - » Mai puține instrucțiuni de scris
- Mentenanța e **mai simplă**
  - » Aceleași motive
- Programele sunt **portabile (ISA independent)**
  - » Conțin puține detalii dependente de hardware
    - – Pot fi folosite cu modificări minore pe alte mașini
  - » Compilatorul traduce la cod mașină specific (ISA)
  - » Programele în asamblare nu sunt portabile (ISA dependent)

Scenarii frecvente de utilizare LA

- compilatoarele traduc codul sursă in cod maşină
  - îndepărtare de limbajul de asamblare, dar nu de renunțare la el
  - mediile de dezvoltare prezintă facilități de inserare de linii scrise direct în limbaj de asamblare
- componente critice ale SO realizate în LA
  - cât mai puțin timp şi, cât mai puțină memorie
  - nu există funcționalități high level pentru:
    - întreruperi, I/O,
    - procesorul în mod privilegiat

De ce se folosește limbajul de asamblare?

- Acces la hardware, control
  - Doar o parte a aplicației este în asamblare
  - Progamare mixed-mode

- Soft de sistem care necesită acces la hardware
  - » Asambloare, linkeditoare, compilatoare
  - » Interfețe de rețea, drivere diverse
  - » Jocuri video

De ce se folosește limbajul de asamblare? (2)

- Eficiență în spațiu
  - Codul asamblat este compact
- Eficiență în timp
  - Codul asamblat este adesea mai rapid
    - » … codul bine scris este mai rapid
    - » E ușor de scris un program mai lent decât echivalentul în limbaj de nivel înalt
- Eficiența în memorie
  - Nu este critică pentru majoritatea aplicațiilor
  - Codul compact este uneori important
    - – Portabile, IOT, senzori, microcontrolere
    - – Software de control în spațiu

Dar ...

- Se cam ocupă compilatorul de optimizări
- Compilator știe mai bine
- Foarte puține secvențe de cod sunt scrise în limbaj de asamblare
- Merită să știi limbaj de asamblare?

De ce merită limbajul de asamblare?

- Rar vei scrie, mai adesea vei citi
- Vrei să înțelegi ce se întâmplă
- Scrii aplicații high-level, dar nu ești lipsit de înțelegerea detaliilor low-level
- Piloții buni sunt mecanici buni.
- Înțelegând sistemul, interfața hardware-software
  - Devii programator mai bun
  - Înțelegi mai bine aplicații complexe
  - Faci aplicații mai performante, mai robuste
  - Depanezi mai uso

Hello World!

```
section .data
        msg db 'Hello, world!', 0xa
        len dd $ - msg

section .text
        global main
main:
        mov ebp, esp
        mov eax, 4
        mov ebx, 1
        mov ecx, msg
        mov edx, [len]
        int 0x80   ; write(1, msg, len)
        xor eax, eax  ; return 0
        ret
```

Demo

$ git clone https://github.com/iocla/demo.git

$ cd demo/curs-01

$ ls

$ #Demo 1: hello world

$ #Demo 2: inline assembly

$ #Demo 3: textbook example

ARHITECTURA SISTEMULUI DE CALCUL

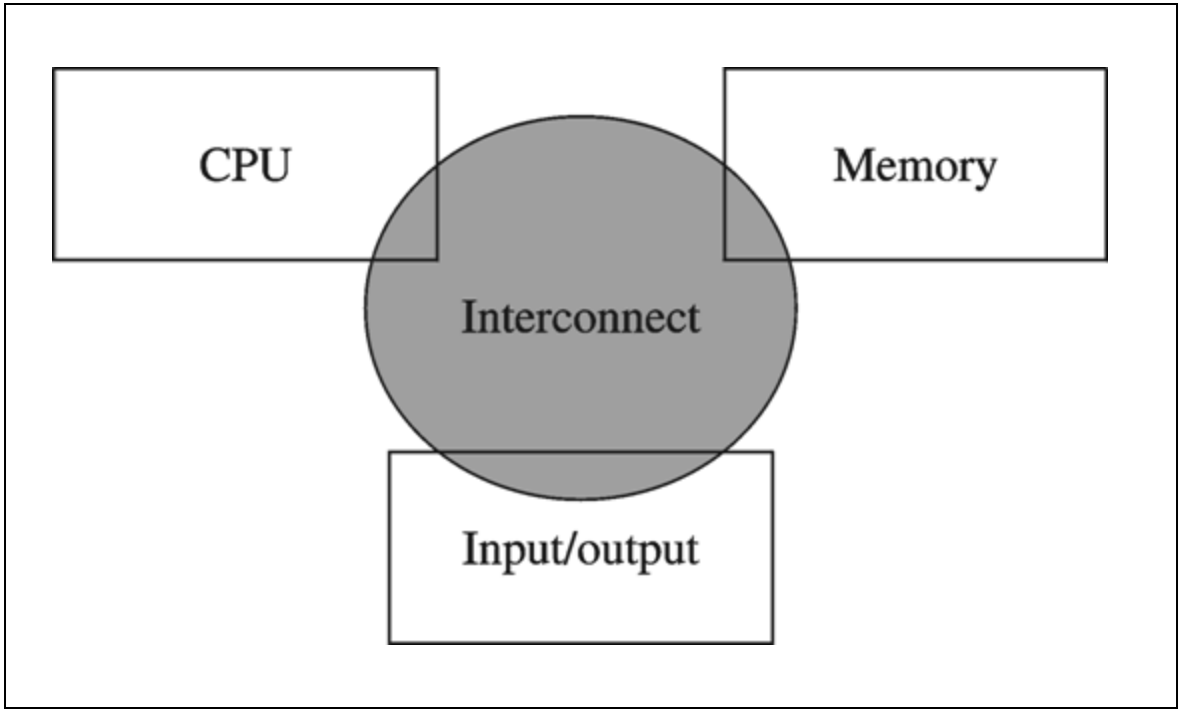
Arhitectura sistemelor de calcul

- 3 semestre: CN1, CN2, ASC
- Componentele unui sistem de calcul
- Funcționarea procesorului
- Funcționarea memoriei
- Input/Output

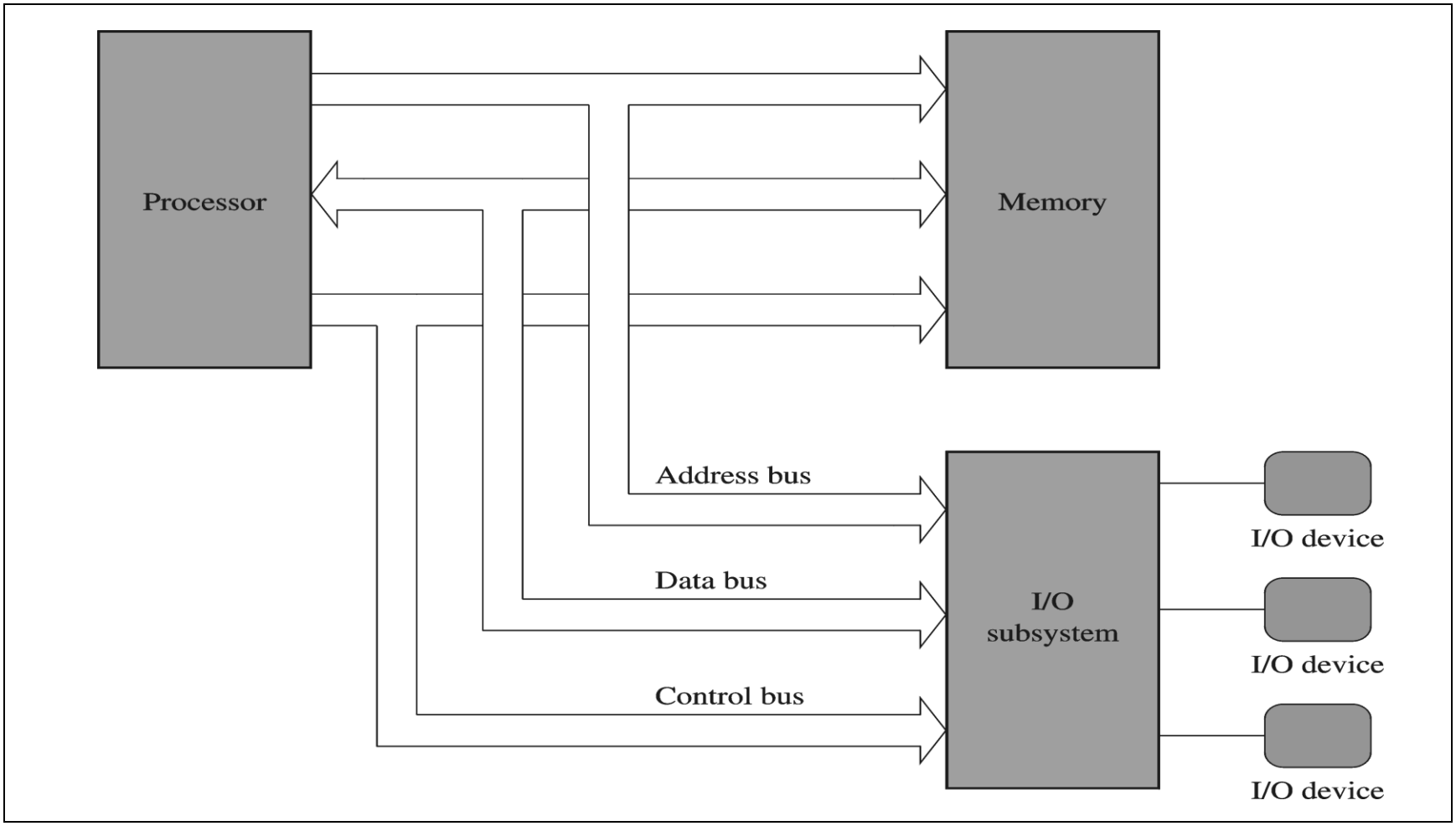
De citit:
Capitolul 2: 2.1, 2.2 2.5 2.6

Componentele de bază

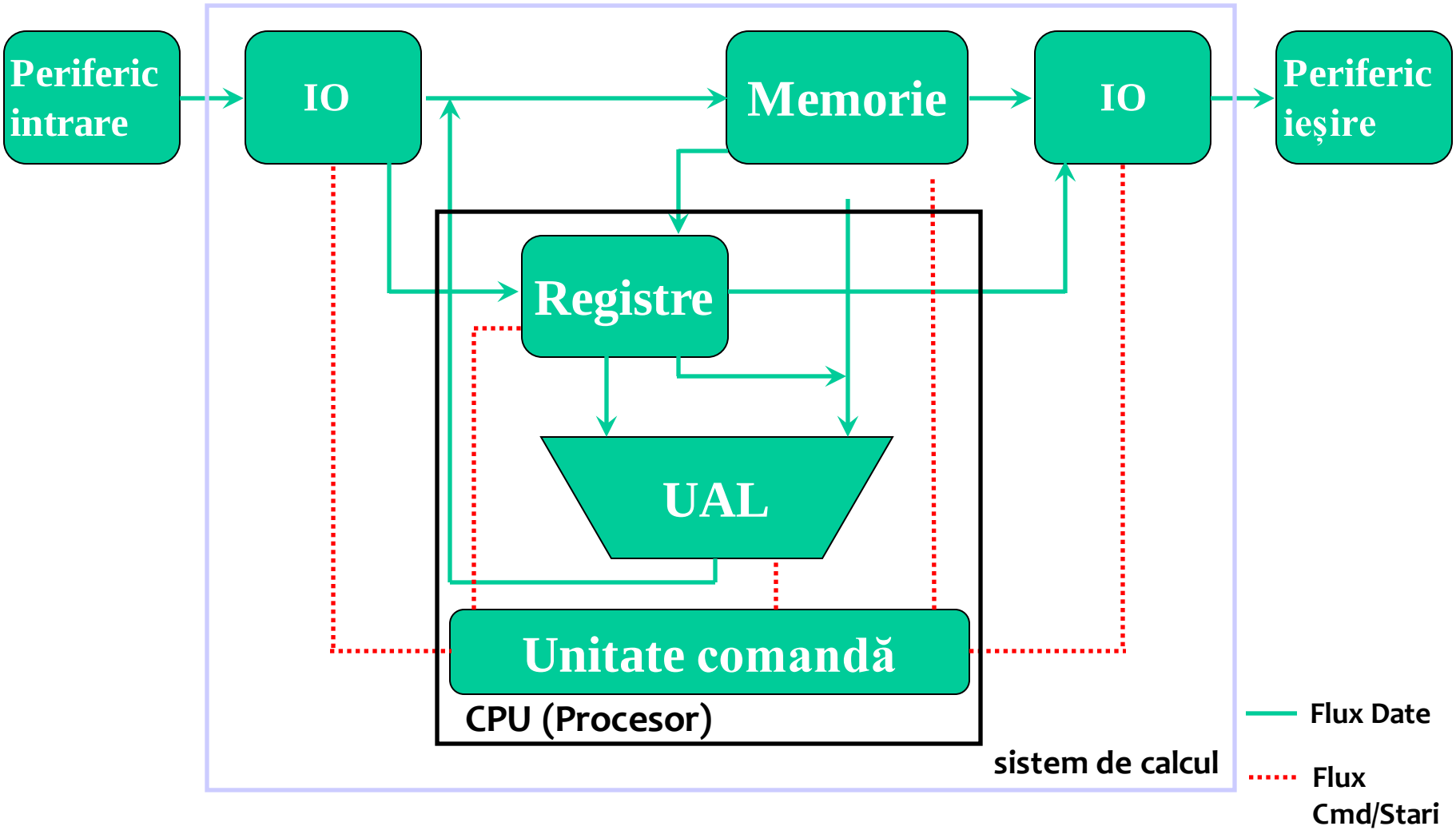
- Sistem de calcul
  - Procesor
  - Memorie
  - Sistem I/O
  - Magistrale (Bus)
    - » Adrese
    - » Date
    - » Control



Arhitectura von Neumann



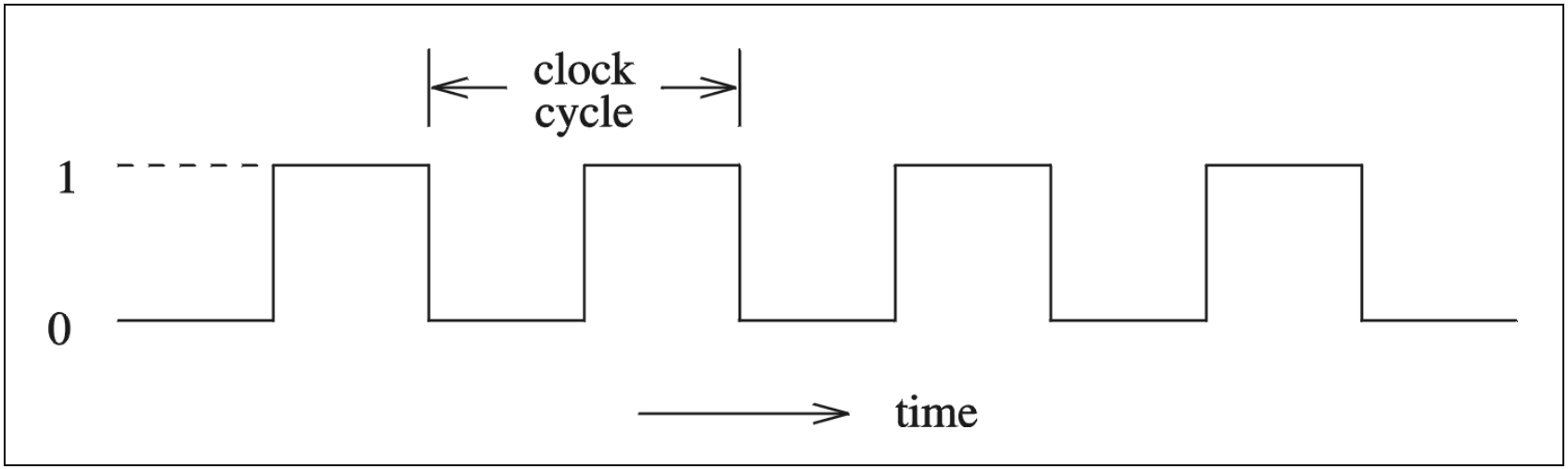
Arhitectura von Neumann



FUNCȚIONAREA PROCESORULUI

Frecvența de lucru

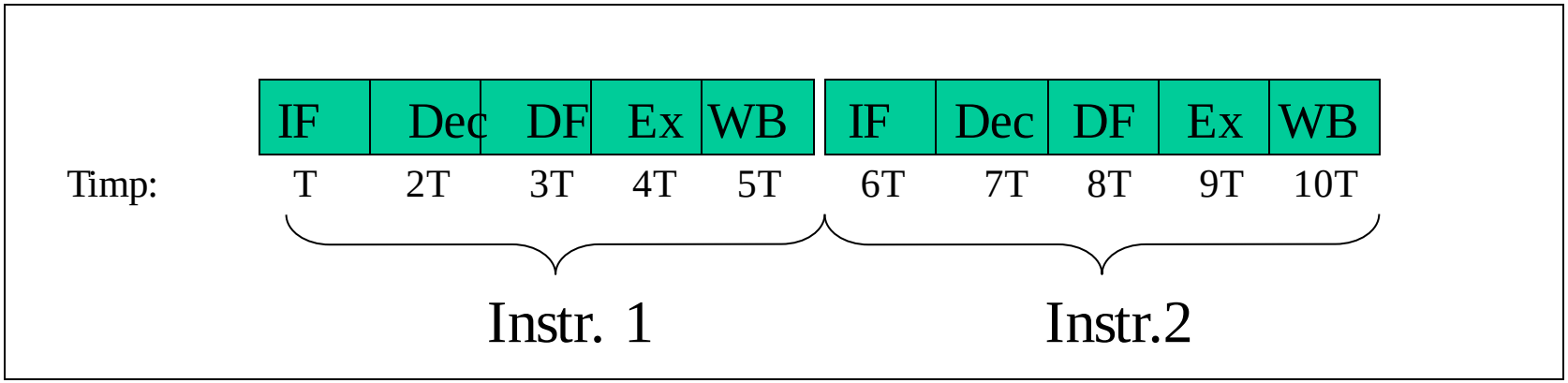
- Ceasul sistemului
  - Semnal de timp
  - perioada $= \frac{1}{\text{Frecvența ceasului}}$



Ciclul de lucru al procesorului

Execută **continuu** bucla:

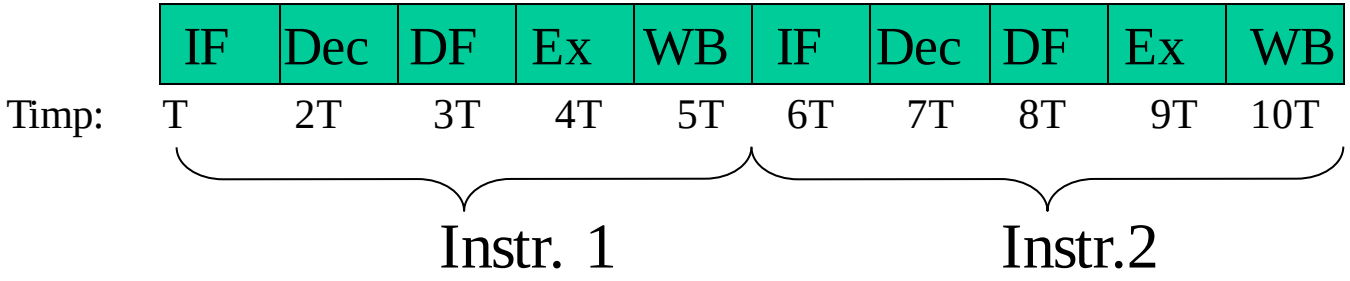
* fetch—decode—data fetch—execute—write back
  - » Fetch – aduce instrucțiunea (cod mașină) din memorie
  - » Decodează instrucțiunea
  - » Aduce date din memorie (dacă e necesar)
  - » Execută instrucțiunea
  - » Actualizează în memorie (dacă e necesar)



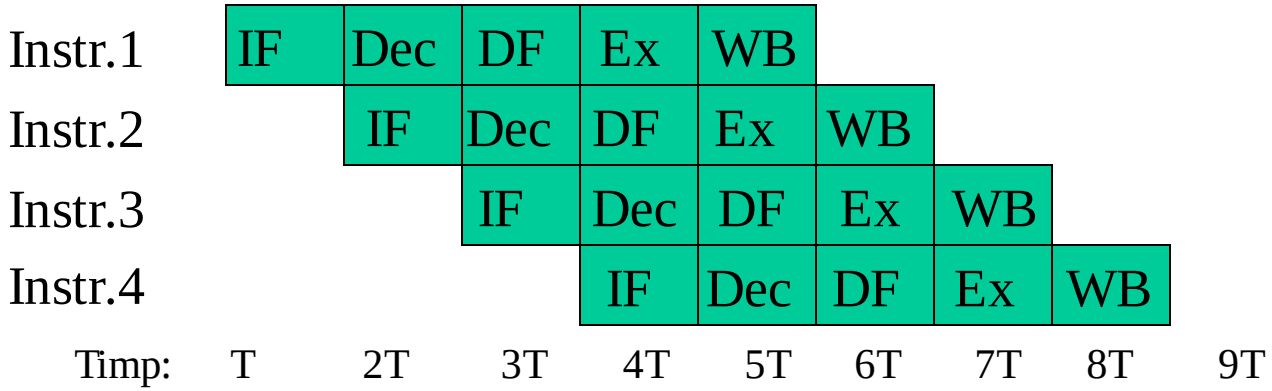
- » În carte: doar 3 pași -- fetch-decode-execute

Execuție secvențială vs pipeline

- Execuție secvențială



- Execuție pipeline



Procesoare RISC vs CISC

- RISC (Reduced Instruction Set Computer)
  - » set redus de instructiuni, multe registre
  - » operanzii sunt registre
  - » instrucțiuni simple: aritmetico-logice, comparații, salturi
  - » doar load/store cu memoria
  - » n++ în MIPS:

```
ldw r1, $n
add r1,1,r1
stw $n, r1
```

- CISC (Complex Instruction Set Computer)
  - » puține registre
  - » instrucțiuni mai complexe
  - » multe instrucțiuni cu operanzi în memorie
  - » o instrucțiune de pe un procesor CISC se poate descompune într-o suită de instrucțiuni RISC
  - » n++ în x86:

```
inc [n]
```

Procesoare RISC vs CISC

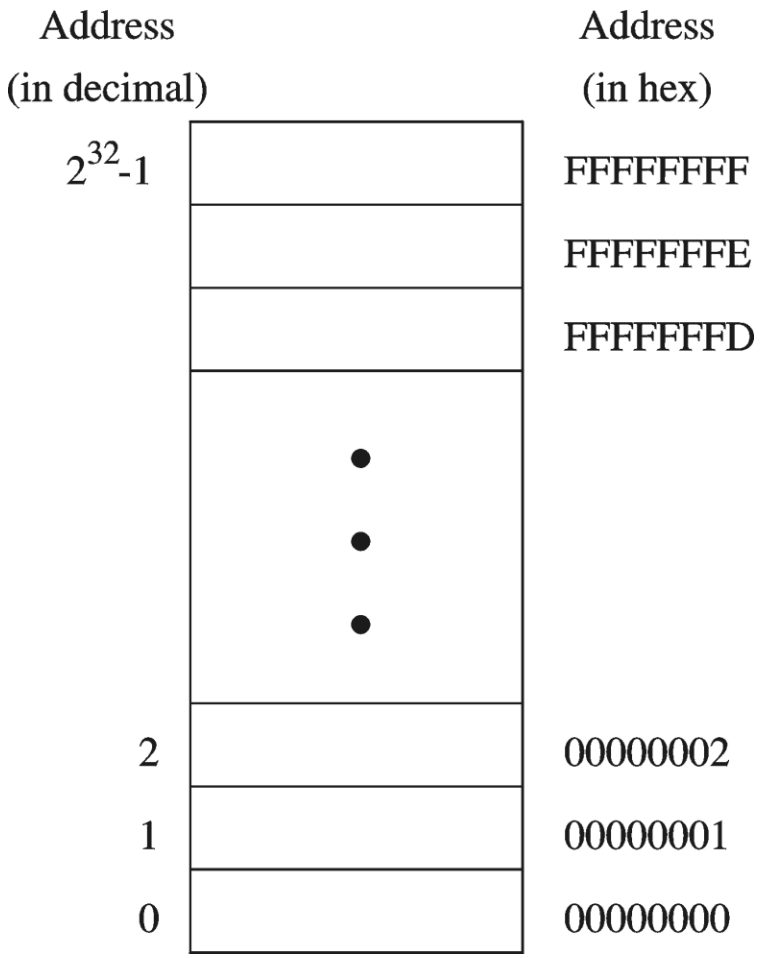
- RISC
  - » Hardware mai ușor de realizat
  - » Durata instrucțiunilor este relativ egală
  - » Lungimea instrucțiunilor este egală
  - » Codul mașină generat de compilator este mai mare
  - » Exemple: ARM, MIPS, Atmel, POWER7

- CISC
  - » Durata de execuție mai mică pentru operațiile frecvente
  - » Hardware mai complicat
  - » Pot exista instrucțiuni foarte scurte, sau foarte lungi
  - » Exemplu: x86, x86_64

FUNCȚIONAREA MEMORIEI

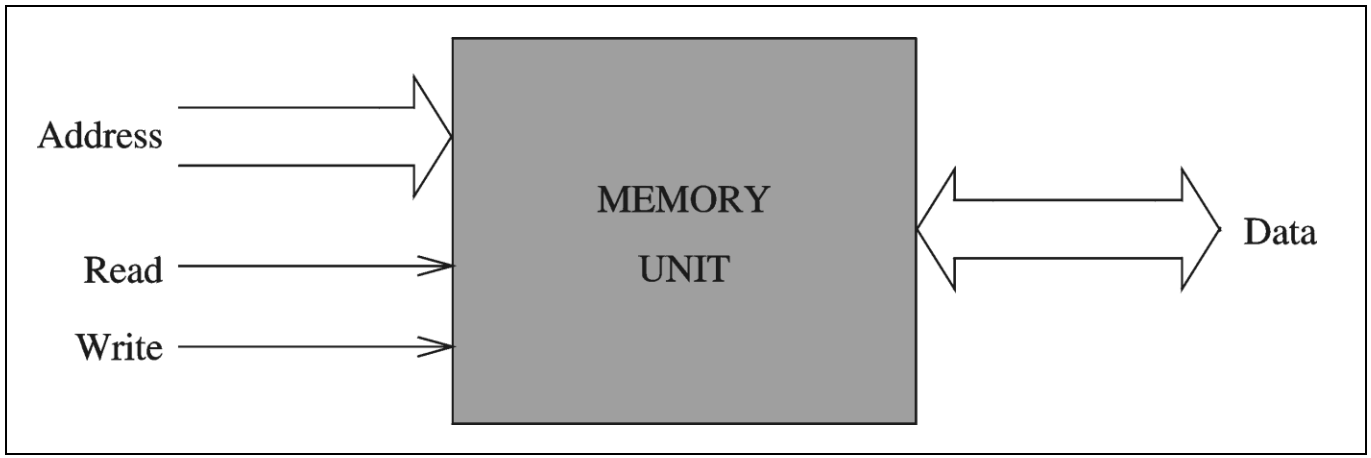
Memoria

- Secvență de octeți
- Fiecare octet are o adresă
  - Adresa este numărul de secvență al octetului
  - mărimea spațiului de adrese
- Adrese logice: fiecare program are adrese 0-FFFFFFFF
- Translatare adrese logice <=> fizice
  - cursuri SO, CN



Memoria

- Două operații de bază
  - Read (citire)
  - Write (scriere)
- clock speed: frecvența de operare
- transfer rate: clock speed * word size / 8
- exemplu PC3-12800: cycle time=5ns, 12800MB/s



Cum scrie/citește CPU în memorie?

Citirea datelor din RAM:

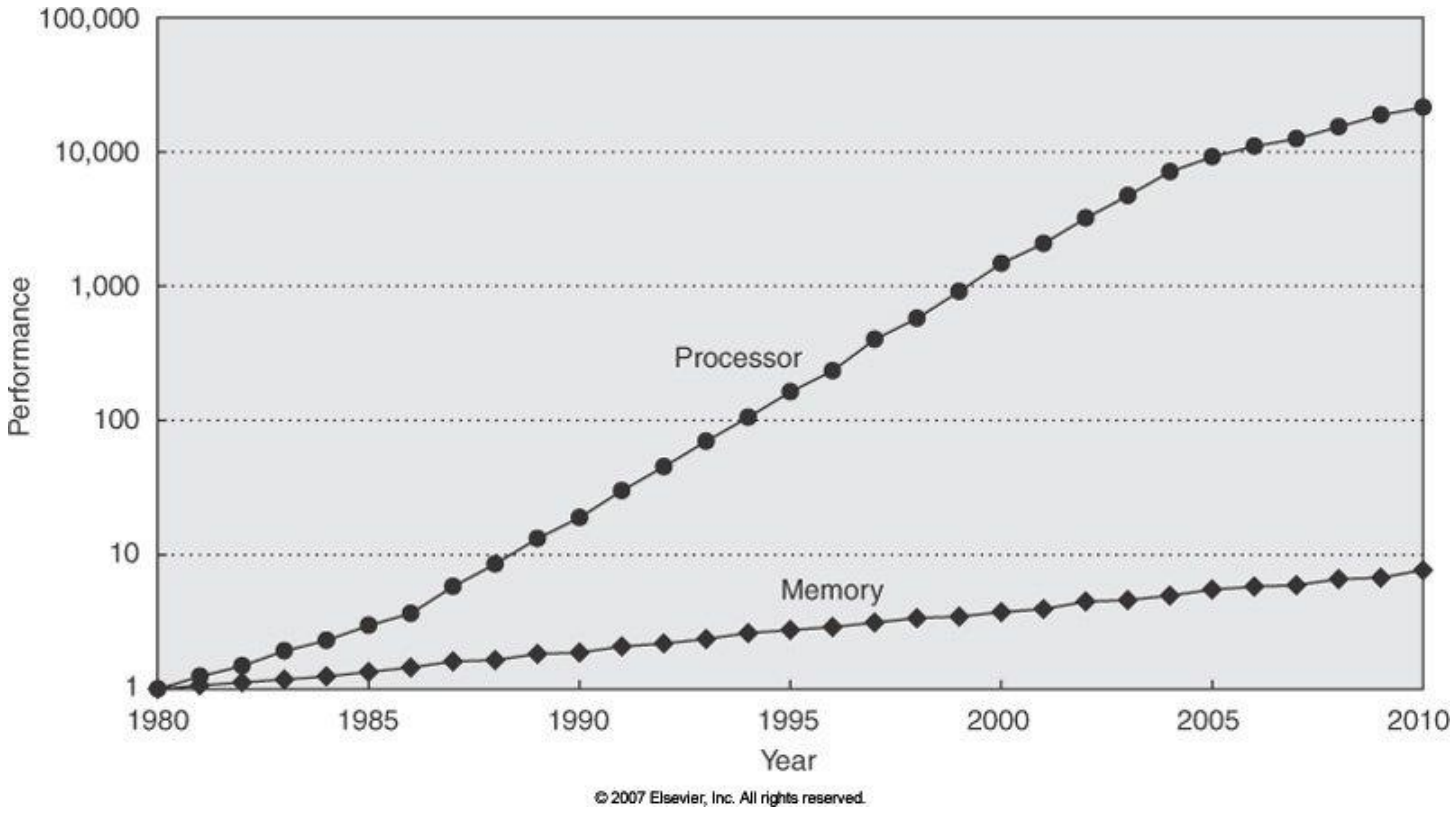
1. pune semnalul de **citire de date** pe Control Bus
2. pune adresa datelor solicitate pe Address Bus
3. citește datele de pe Data Bus și le pune intr-un registru

Scrierea datelor în RAM:

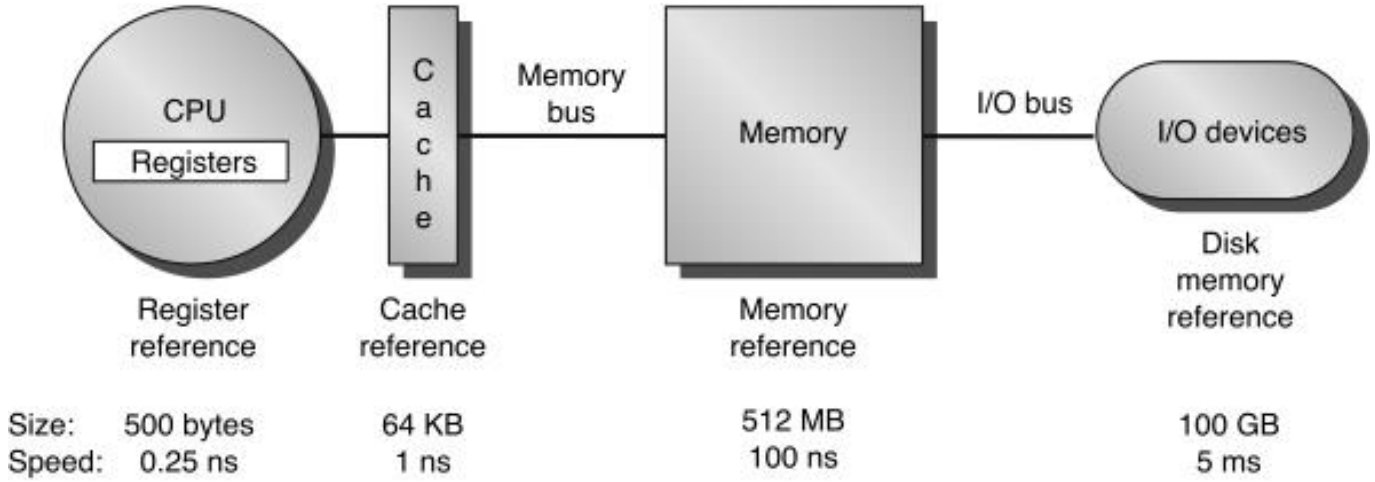
1. pune semnalul de **scriere de date** pe Control Bus
2. pune adresa datelor dorite pe Address Bus
3. pune datele de scris pe Data Bus
4. așteaptă ca memoria RAM să execute efectiv scrierea

Istoric performanțe memorie vs. CPU

* O citire la x86 durează ~ 3 cicli
* memoria este azi e mult mai lentă decât procesorul



Tipuri de memorii si latențele lor



Viteza: **mare** → **mică**

Capacitate: **mică** → **mare**

Caracteristici tipice ale memoriilor

	Dimensiune	Latența (ns)	Lațime de bandă (MB/sec)	Gestionat de
Registre	Octeți	0.25		compilator
Cache	~16MB	0.5 (L1) - 7 (L2)	200,000	hardware
Memoria RAM Rețea infiniband	~ 16GB	100 3000	10,000 3500	SO software/SO
Discuri	~ 4TB	5,000,000	200	SO

Cuvinte cheie

- sistem de calcul
- von Neumann
- procesor
- memorie
- I/O
- magistrală
- CISC
- RISC
- pipeline
- adresă
- citire și scriere

Intrebări?