Introducere în Organizarea Calculatoarelor și Limbaj de Asamblare

Modificat: 24-Sep-18

Echipa

Laborator: Răzvan Nițu, Claudiu Ghioc, Ebru Resul, Robert Baronescu, Cristi Baciu, Ionuț Mihalache

Teme: Ioana Ciornei, Vladimir Diaconescu, Tiberiu Lepădatu, Radu Nicolau, Andrei Preda, Mădălina Moga, Vlad Vitan, Bogdan Firuți

Curs: Dragoș Niculescu, Voichița Iancu, Costin Boiangiu, Dan Novischi, Adriana Cogean, Cristian Crețeanu, Teodor Apostol

Lucrări de curs: Bogdan Purcăreață, Neculai Balaban, Răzvan Deaconescu, Alexandra Pîrvulescu, Andra Danciu, Ștefan Brătescu, Diana Grecu

Comunitate: Daniel Băluță, Radu Velea

Infrastructură: Dănuț Matei

Coordonator: Răzvan Deaconescu

Resurse curs

- Instanță de pe cs.curs.pub.ro
- Wiki: http://ocw.cs.pub.ro/iocla
- Sala de laborator EG410A
- Săli de curs: EC004, EC105
- Textbook: Sivarama P. Dandamudi *Introduction to Assembly Language Programming For Pentium and RISC Processors, 2nd Edition*, Springer 2005

Toate indicațiile "de citit" capitole/anexe se referă la această carte.

Bibliografie extinsă

OBLIGATORIU

- **Sivarama P. Dandamudi – "Introduction to Assembly Language Programming For Pentium and RISC Processors", Springer, 2005**
- Ray Sefarth, "Introduction to 64 Bit Intel Assembly Language Programming for Linux", 2011, cap 16 (optimizări)
- Richard Blum, "Professional Assembly Language" , Wiley 2005, cap 15 (optimizări)

SUPLIMENTAR

- Kip R. Irvine - Assembly Language for x86 Processors (7th Edition), Pearson, 2015 - Windows, MASM, VisualC
- Jeff Dunteman - Assembly Language Step By Step, 3rd Edition. Wiley, 2009, Linux, NASM

- NU învățați după slide-uri, cartea este **OBLIGATORIE**

Cuprins tentativ

- Introducere
  - De ce? Locul cursului în CS, Hello World
- Arhitectura sistemelor de calcul
- Arhitectura X86
  - Procesor, bus, memorii, SO, intreruperi
- Reprezentarea datelor în sistemele de calcul
  - Bin, Hex, Complement față de 2
- Setul de instrucțiuni
  - aritmetice, logice, control flux, șiruri de caractere
- Declarare, adresare
- Unelte, scule, utilitare
- Stiva, funcții, C + assembler
- Buffer overflow, securitate
- Optimizări

Sistem de notare

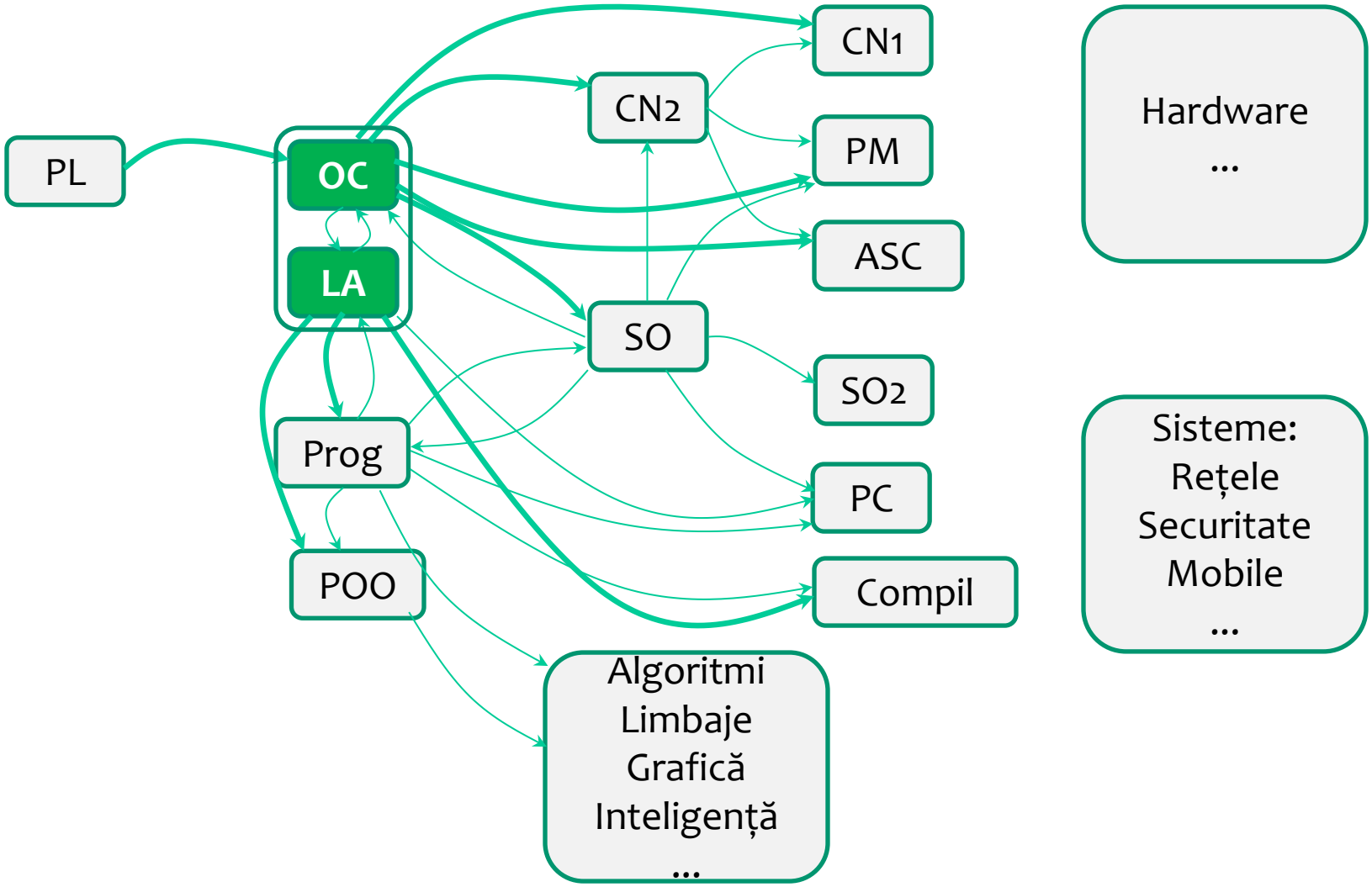
- **[60%] Curs**
  - [20%] 4 evaluări de 15min în timpul semestrului
  - [40%] examen practic
- **[40%] Laborator**
  - [10%] Activitate de laborator (12 laboratoare)
  - [30%] Teme de casă (3 teme)

- **Atenție: Este necesară promovarea independentă curs + laborator!**

Calendar curs

1.	25-Sep-2018	1-Oct-2018	Curs 01 - intro + ASC
2.	2-Oct-2018	8-Oct-2018	Curs 02 - Arhitectura x86
3.	9-Oct-2018	15-Oct-2018	Curs 03 - reprezentare date
4.	16-Oct-2018	22-Oct-2018	Curs 04 - **L1**, set de instructiuni
5.	23-Oct-2018	29-Oct-2018	Curs 05 - adresare, declarare
6.	30-Oct-2018	5-Nov-2018	Curs 06 - Unelte, utilitare
7.	6-Nov-2018	12-Nov-2018	Curs 07 - **L2** stiva f() c/asm
8.	13-Nov-2018	19-Nov-2018	Curs 08 - stiva f() c/asm
9.	20-Nov-2018	26-Nov-2018	Curs 09 - stiva f() c/asm
10.	27-Nov-2018	3-Dec-2018	Curs 10 - **L3**, Buffer overflows
11.	4-Dec-2018	10-Dec-2018	Curs 11 - Buffer overflows
12.	11-Dec-2018	7-Jan-2019	Curs 12 - optimizări
13.	8-Jan-2019	15-Jan-2019	Curs 13 - **L4** recapitulare

Importanța IOCLA în cs.pub.ro



Cuprins curs 1

De citit:
Cap 1, Anexa B

- 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?
- Hello World
- Arhitectura Sistemelor de Calcul

De citit:
Capitolul 2: Fără 2.3, 2.4 și 2.7

Cum se vede sistemul la utilizator

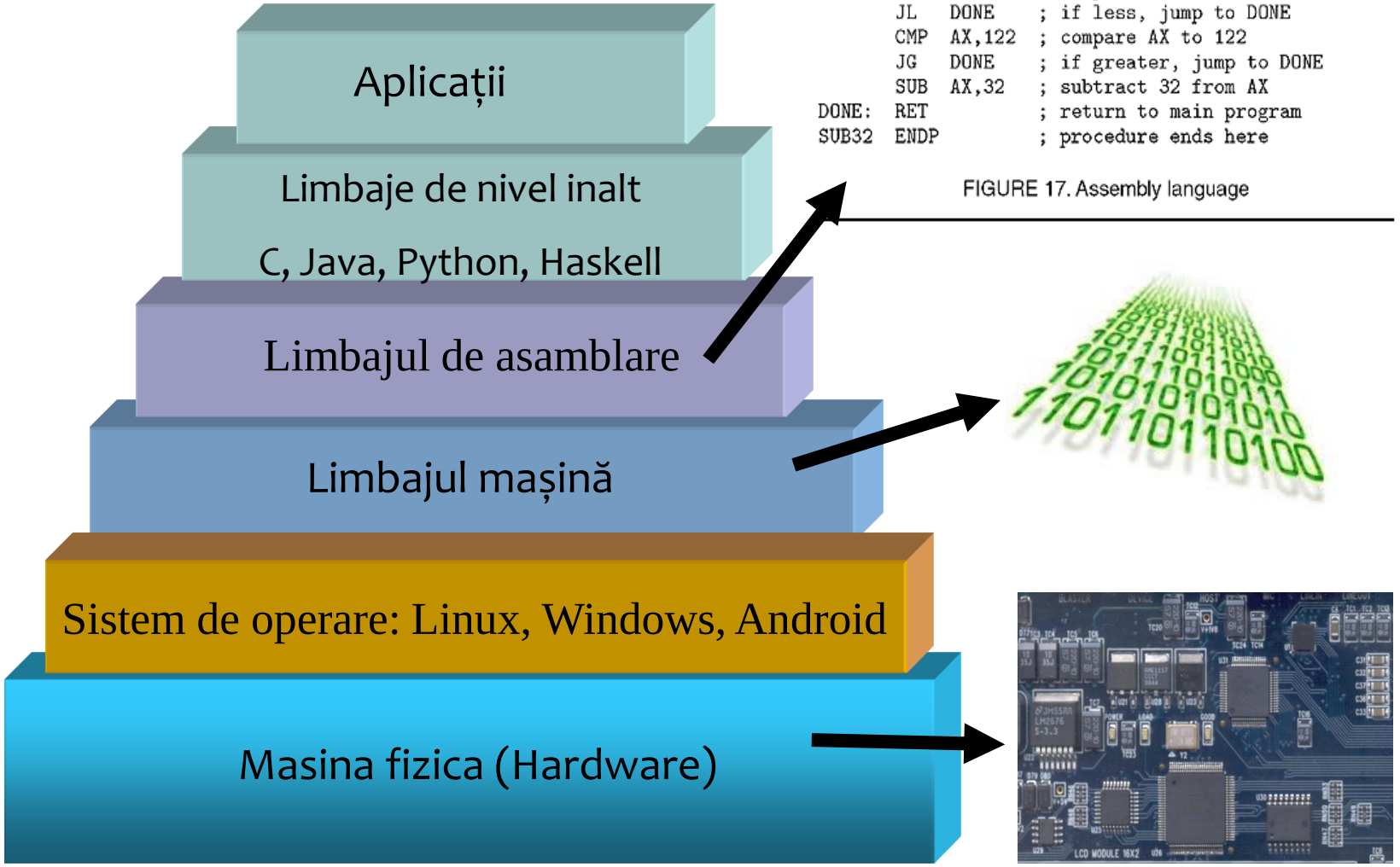
- Depinde de gradul de abstractizare software
- Ierarhie de 6 nivele
  - Vârful ierarhiei izolează utilizatorul de hardware
  - *Idependente de sistem:* primele două
  - *dependente de sistem:* ultimele trei
    - » Limbajele asamblare/mașină sunt specifice procesorului
    - » Corespondență directă 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



Ce este limbajul de asamblare?

- Limbaj de nivel jos
  - » Fiecare instrucțiune rezolvă un task simplu
- Corespondență directă între limbajele 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)

Ce este limbajul 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
```

Ce este limbajul de asamblare?

- 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;`

Ce este limbajul de asamblare?

- 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`

Ce este limbajul de asamblare?

- 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`

Ce este limbajul de asamblare?

- Exemplu MIPS

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

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**
  - » Conțin puține detalii dependente de hardware
    - – Pot fi folosite cu modificări minore pe alte mașini
  - » Compilatorul traduce la cod mașină specific
  - » Programele în asamblare nu sunt portabile

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?

- Aces la hardware, control
  - Doar o parte a aplicației este în asamblare
  - Progamare mixed-mode
- Eficiența în spațiu
  - Codul asamblat este compact
- Eficiența 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

De ce se folosește limbajul de asamblare?

- Specialiștii care se respectă știu (și) limbaj de asamblare
- Înțelegere a modului in care lucreaza un calculator
- Programare sisteme incapsulate (embedded)
- Scrierea unor programe eficiente (timp&spațiu)
- Optimizări de timp rulare
- Opimizări de spațiu de memorie ocupat
- Înțelegerea, proiectarea și implementarea securității aplicațiilor
- Dorința de a încerca ceva nou

De ce se folosește limbajul de asamblare?

- Aces la hardware
  - Soft de sistem care necesită acces la hardware
    - » Asambloare, linkeditoare, compilatoare
    - » Interfețe de rețea, drivere diverse
    - » Jocuri video
- Eficiența în memorie
  - Nu este critică pentru majoritatea aplicațiilor
  - Codul compact este uneori important
    - – Portabile, IOT, senzori, microcontrolere
    - – Software de control în spațiu

Arhitectura sistemelor de calcul

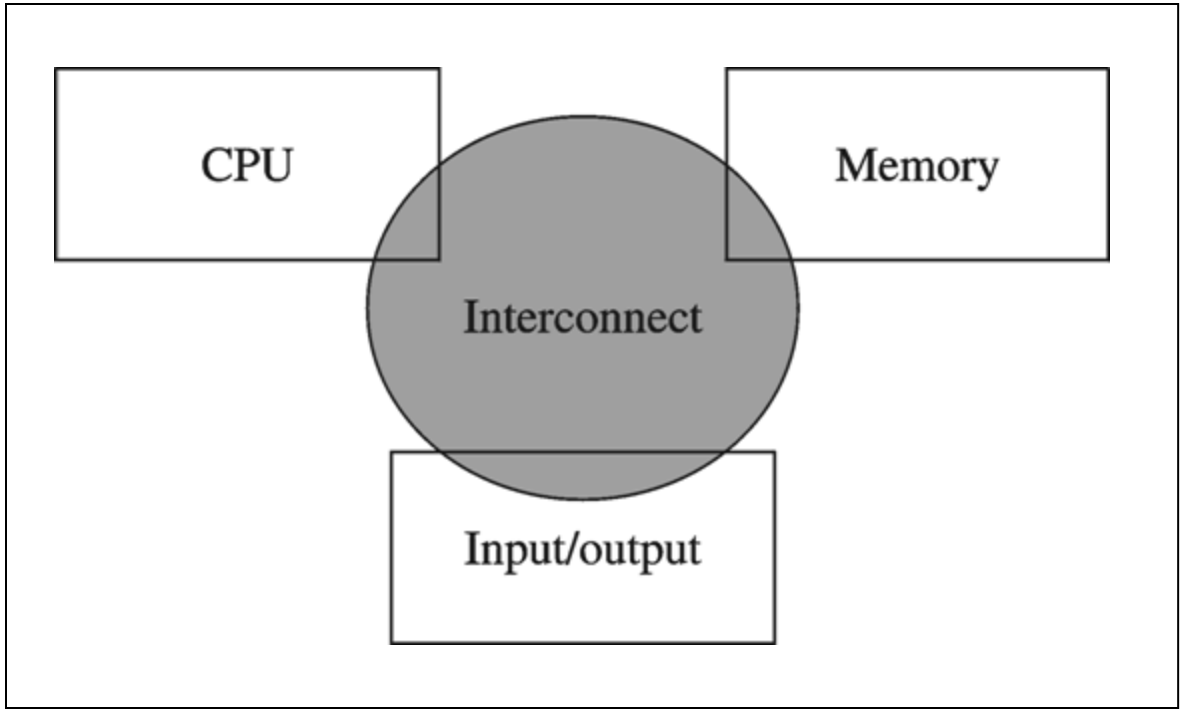
- 3 semestre: CN1, CN2, ASC
- Componentele unui sistem de calcul
- Funcționarea procesorului
- Funcționarea memoriei
- Input/Output

De citit:

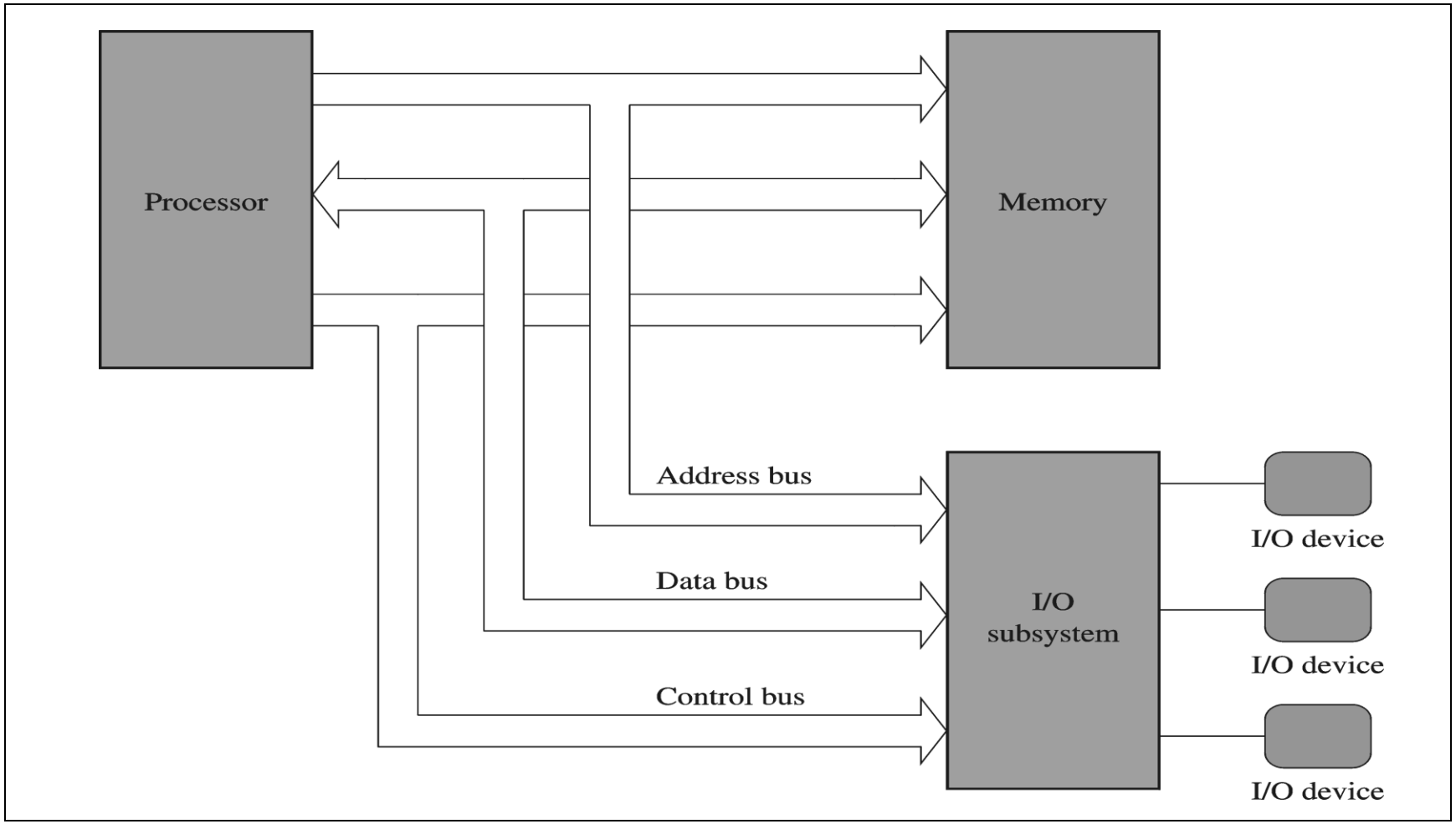
Capitolul 2: Fără 2.3, 2.4 și 2.7

Componentele de bază

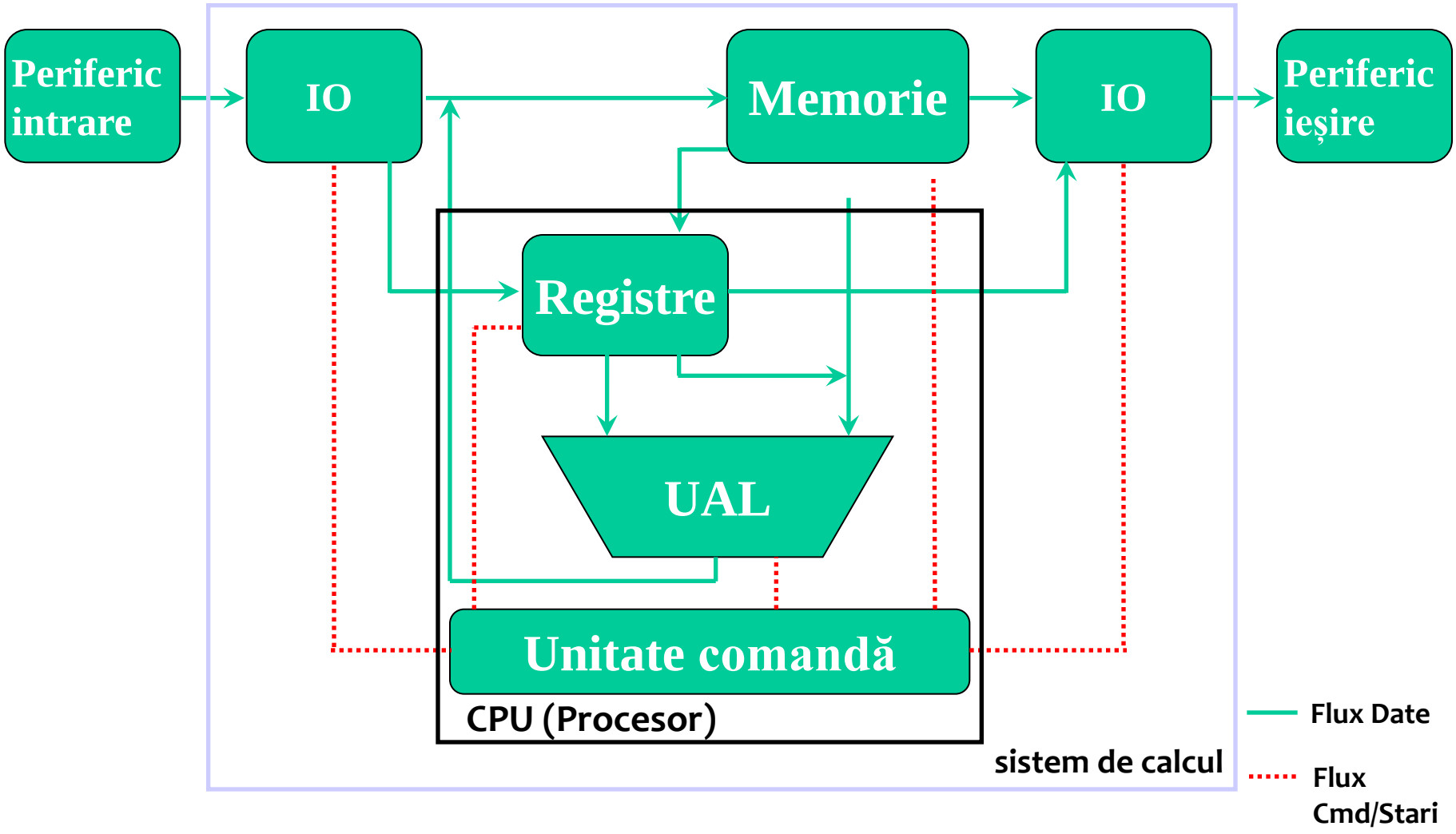
- Sistem de calcul
  - Procesor
  - Memorie
  - Sistem I/O
  - Magistrale (Bus)
    - » Adrese
    - » Date
    - » Control



Arhitectura von Neumann



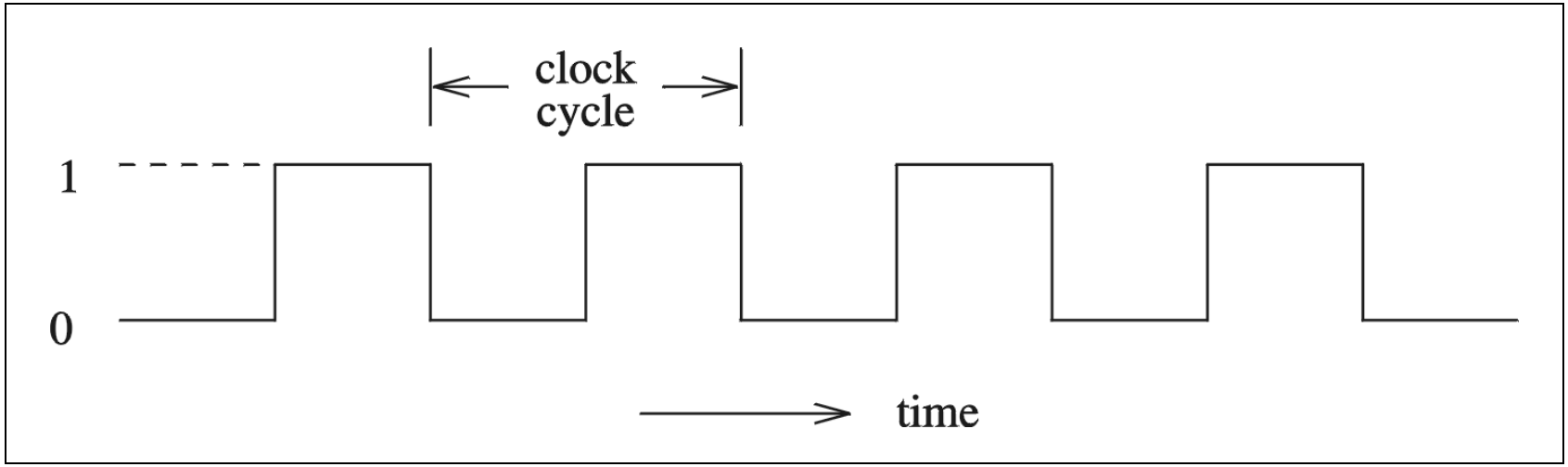
Arhitectura von Neumann



FUNCȚIONAREA PROCESORULUI

Frecvența de lucru

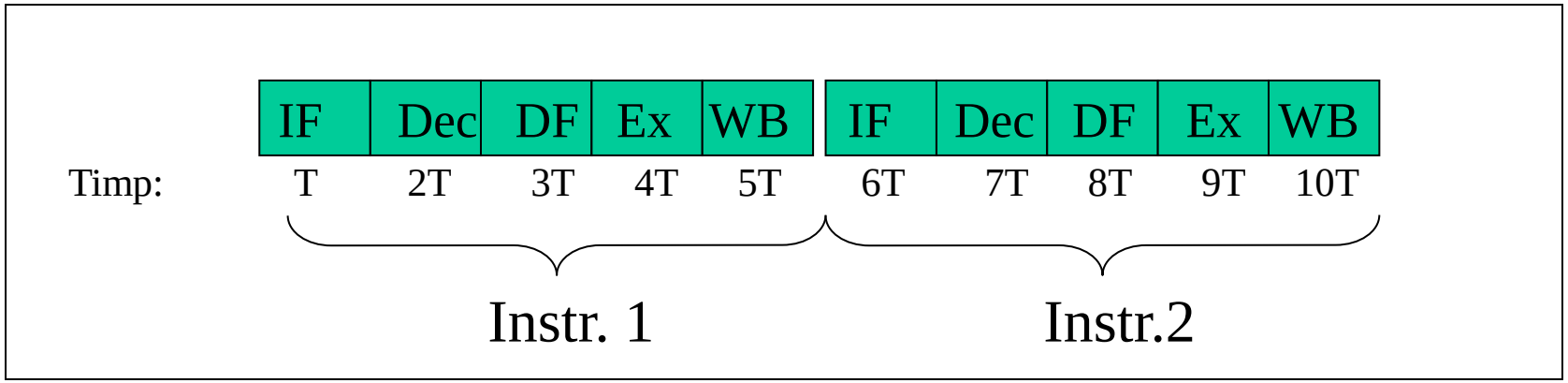
- Ceasul sistemului
  - Semnal de timp
  - perioada $= \frac{1}{\text{Frecvența ceasului}}$



Ciclul de lucru al procesorului

Execută **continuu** bucla:

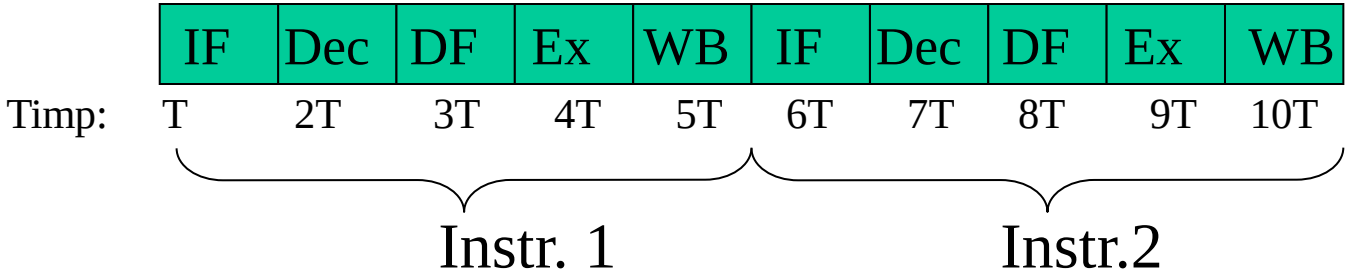
* fetch—decode—data fetch—execute—write back
  - » Fetch – aduce instrucțiunea (cod mașină) din memorie
  - » Decodează instrucțiunea
  - » Aduce date din memorie (dacă e necesar)
  - » Execută instrucțiunea
  - » Actualizează în memorie (dacă e necesar)



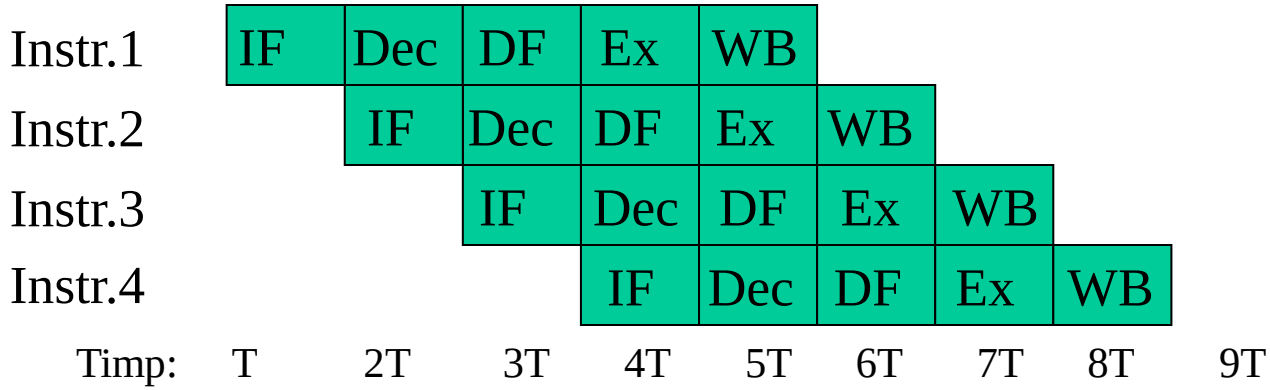
- » În carte: doar 3 pași -- fetch-decode-execute

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

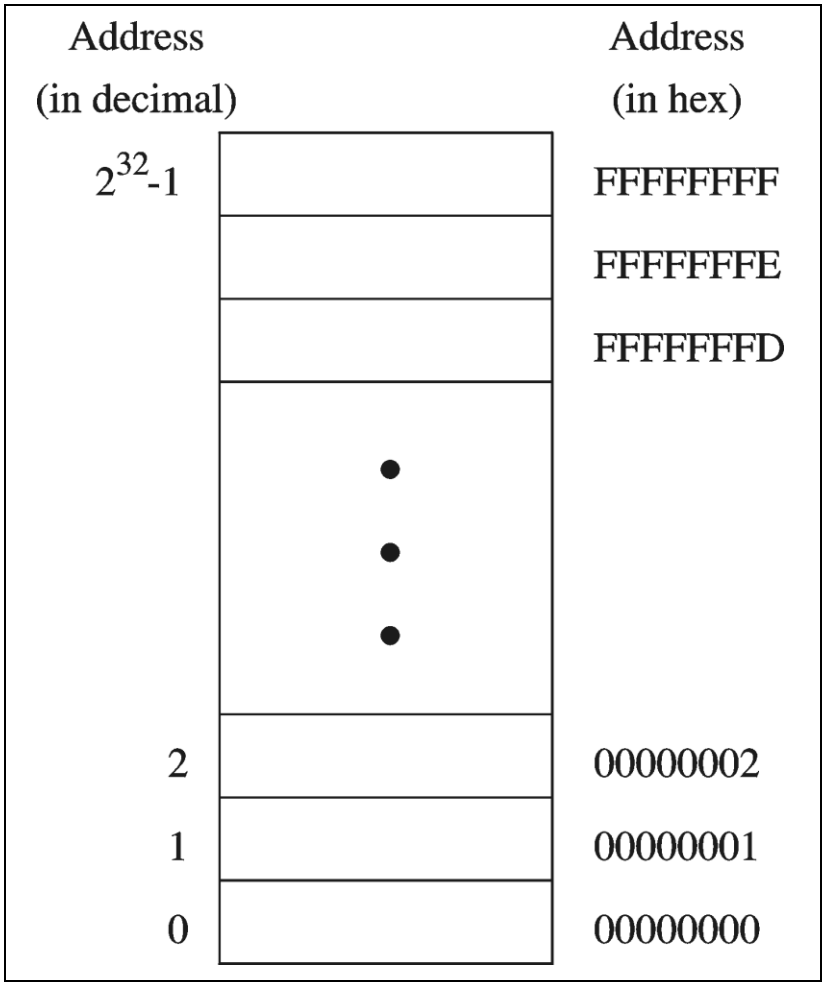
- RISC
  - » Hardware mai ușor de realizat
  - » Durata instrucțiunilor este relativ egală
  - » Lungimea instrucțiunilor este egală
  - » Codul mașină generat de compilator este mai mare
  - » Exemple: ARM, MIPS, Atmel, POWER7

- 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

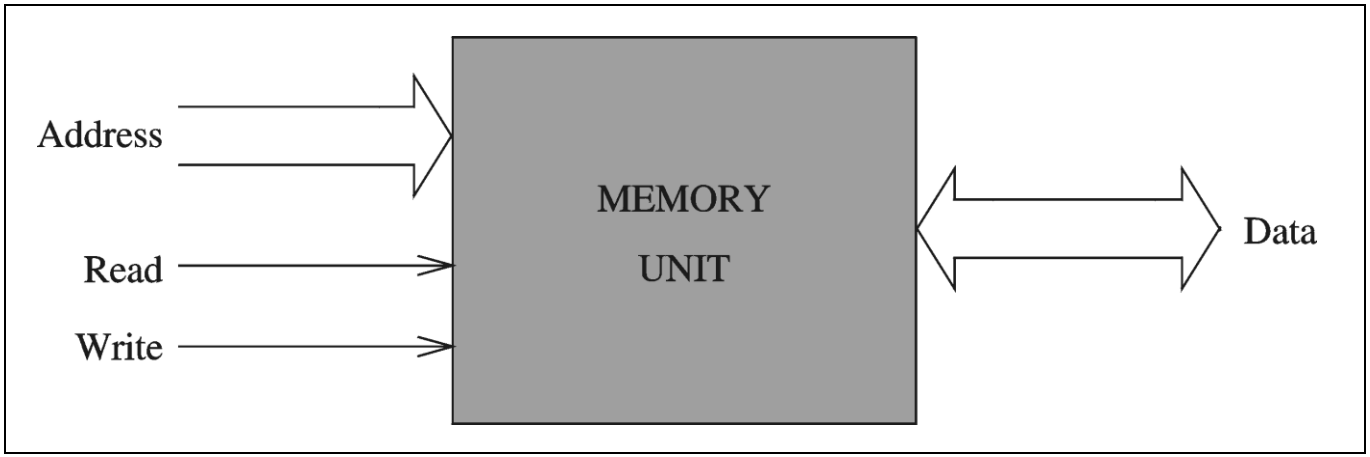
Memoria

- Secvență de octeți
- Fiecare octet are o adresă
  - Adresa este numărul de secvență al octetului
  - *byte addressable (soft)*
  - *word addressable (hard)*
  - mărimea spațiului de adrese
  - Hardware: de fapt se citesc cuvinte de 64 biți (DDR2, DDR3)



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:

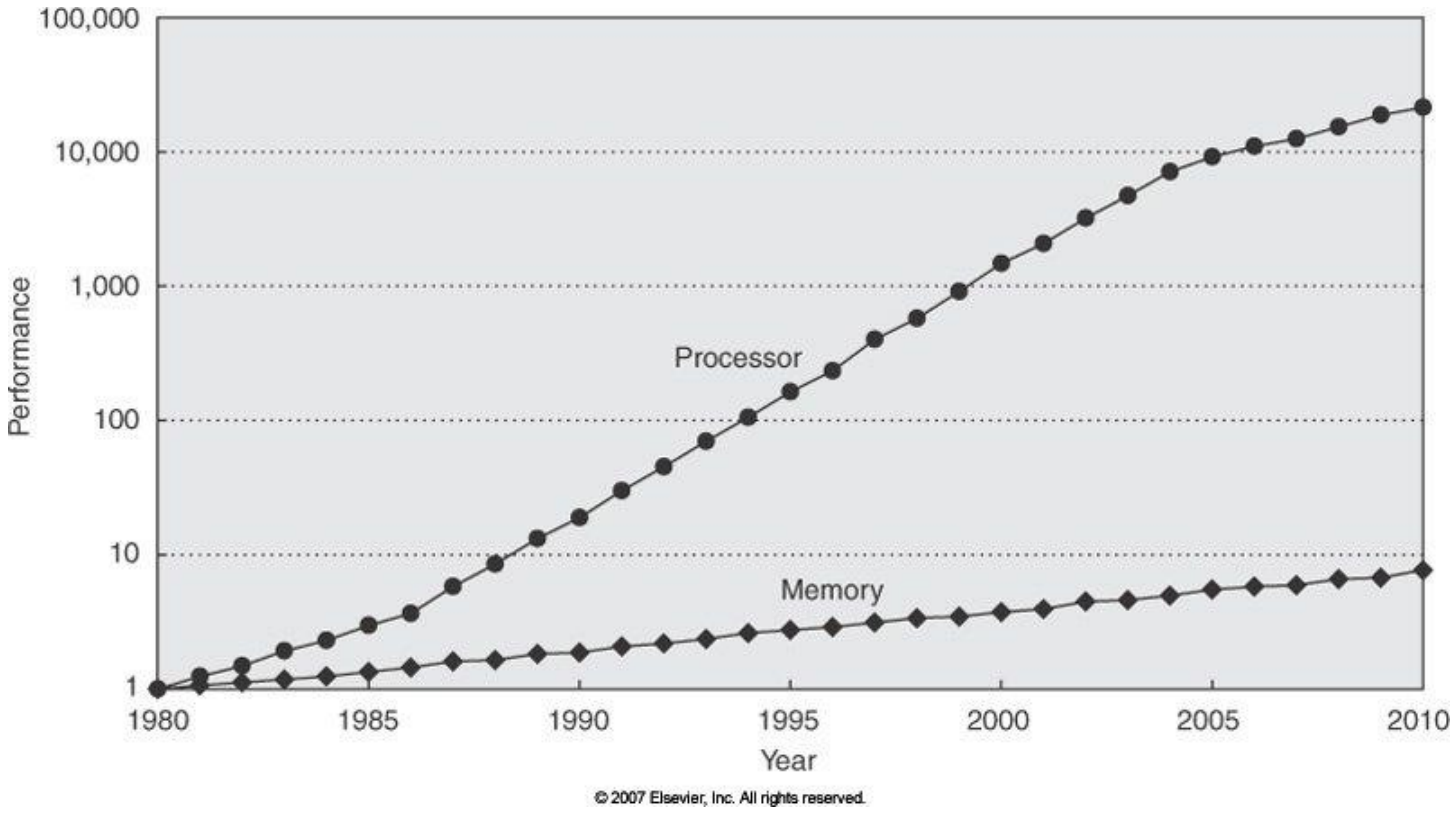
1. pune semnalul de **citire de date** pe Control Bus
2. pune adresa datelor solicitate pe Address Bus
3. citește datele de pe Data Bus și le pune intr-un registru

Scrierea datelor în RAM:

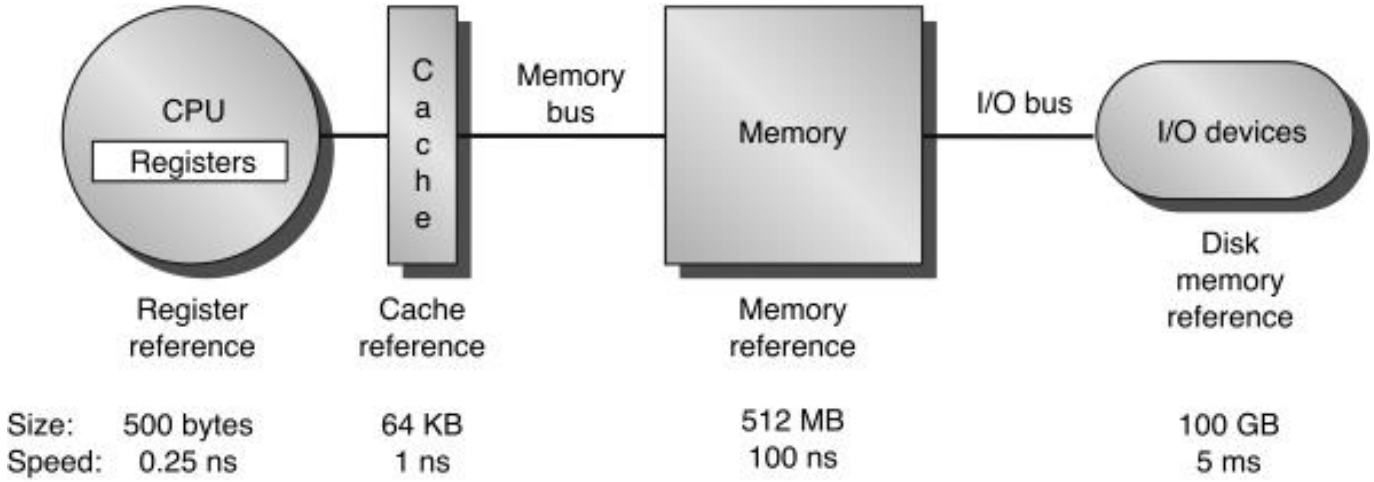
1. pune semnalul de **scriere de date** pe Control Bus
2. pune adresa datelor dorite pe Address Bus
3. pune datele de scris pe Data Bus
4. așteaptă ca memoria RAM să execute efectiv scrierea

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**

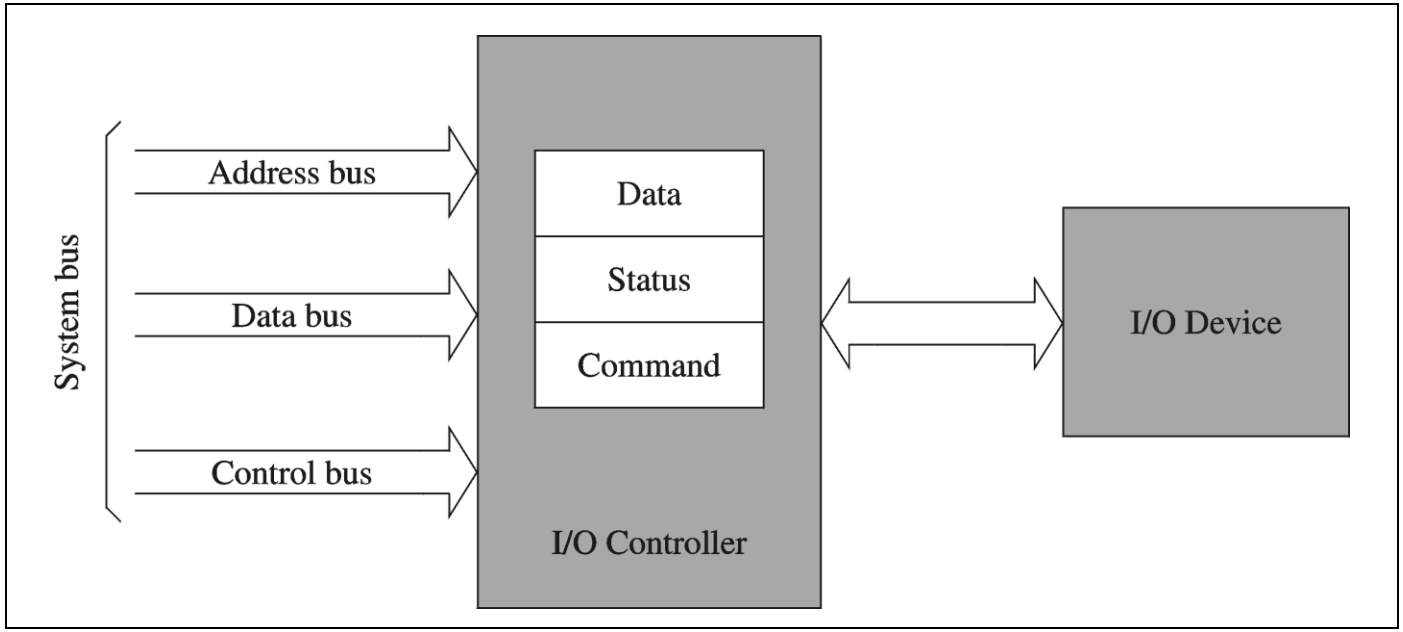
Caracteristici tipice ale memoriilor

	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	~ 16GB	100	10,000	O/S
Rețea infiniband	---	3000	3500	software
Discuri	~ 4TB	5,000,000	200	O/S

INPUT/OUTPUT

Input/Output

- Controller-ul I/O = interfața cu dispozitivele externe
  - Detalii dependente de dispozitiv
  - Interfața electrică necesară



Porturi I/O

- Procesorul comunică cu dispozitivele prin ***porturi***
  - Port: Adresă de I/O
- Un dispozitiv are un set de registre interne accesibile prin adrese de magistrală
  - O adresă = un registru
- Se folosesc magistralele similar cu memoria
- De obicei transferurile I/O se fac în drivere, în kernel

Maparea porturilor

- Modul în care sunt definite porturile (adresele I/O)
- În memorie
  - Citirea scrierea anumitor adrese duce la dispozitive
  - Intrucțiunile normale ale procesorului de acces la memorie
  - Datele nu ajung la RAM ci la un registru de dispozitiv
- Instrucțiuni specializate I/O
  - Spațiu de adrese separat
  - Un semnal de procesor dictează cum este folosită magistrala de adrese
  - Exemplu: instrucțiunile **`in`** and **`out`**

Cuvinte cheie

- sistem de calcul
- von Neumann
- procesor
- memorie
- I/O
- magistrală
- CISC
- RISC
- pipeline
- adresă
- citire și scriere
- aliniere
- Port I/O
- inb, outb

Intrebări?

- Facultative

Hello World!

Bash

```
echo "Hello World"
```

C

```
#include <stdio.h>
int main(void)
{
    printf(“%s\n”, "Hello, world!");
}
```

C++

```
#include …
int main()
{
    std::cout << "Hello, world!";
    return 0;
}
```

Hello World!

Java

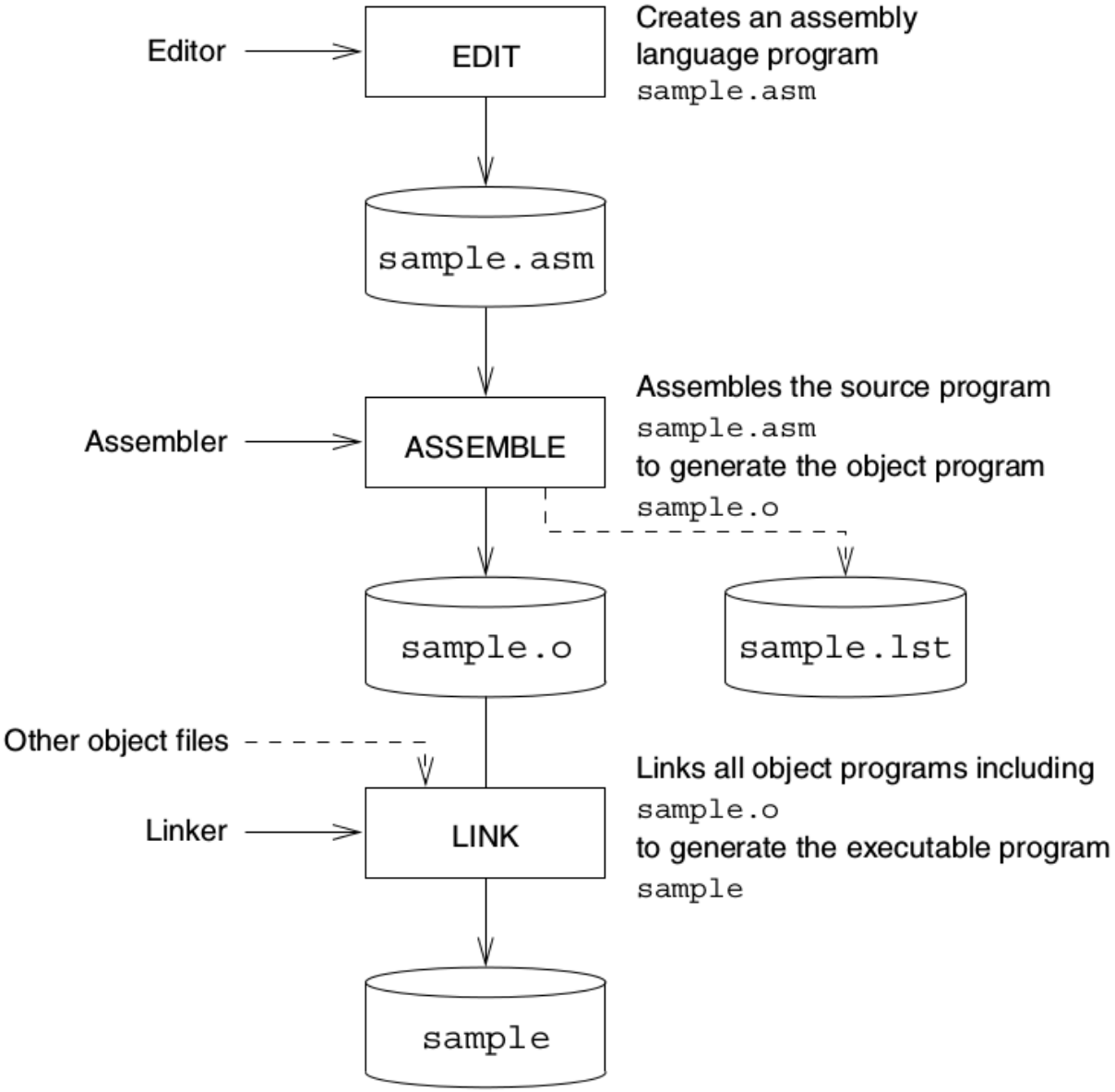
```
import javax.swing.JFrame;  //Importing class JFrame
import javax.swing.JLabel;  //Importing class JLabel
public class HelloWorld {
    public static void main(String[] args) {
        JFrame frame = new JFrame();           //Creating frame
        frame.setTitle("Hi!");                 //Setting title frame
        frame.add(new JLabel("Hello, world!"));//Adding text to frame
        frame.pack();                          //Setting size smallest
        frame.setLocationRelativeTo(null);     //Centering frame
        frame.setVisible(true);                //Showing frame
    }
}
```

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
```

Asamblare, Link editare

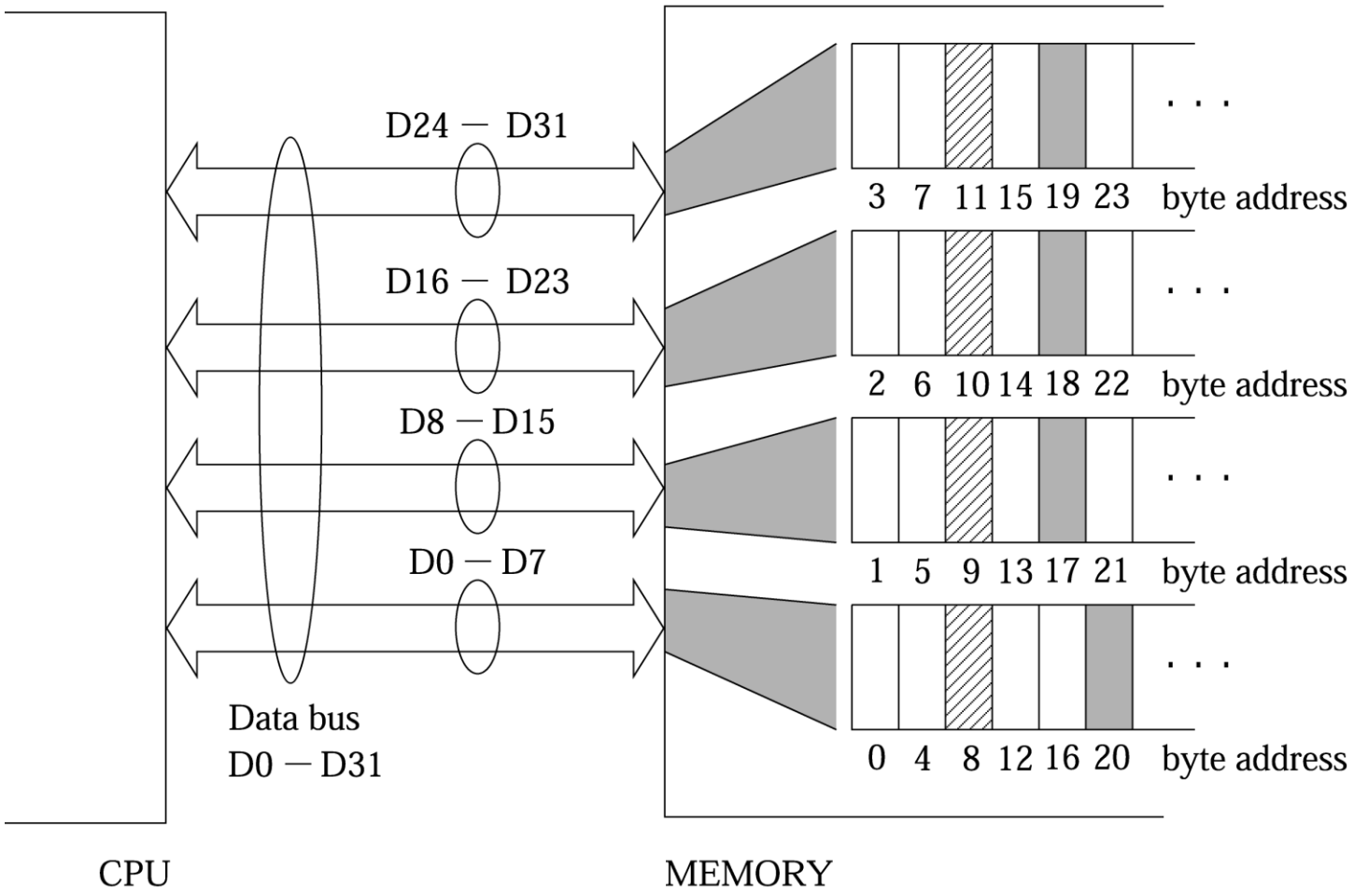


Performanță: alinierea datelor

* Soft alignment
  - » Alinierea nu este obligatorie
  - » Datele aliniate => acces mai rapid
  - » Procesoare Intel x86
* Hard alignment
  - » Alinierea obligatorie
  - » RISC, ARM
* Exemplu
  - » Se citește o variabilă de 32biți
  - » Magistrala de date este de 32 biți
  - » Adresa 8 vs. adresa 17

Performanță: alinierea datelor

Citire **`dword ptr [8]`** vs **`dword ptr [17]`**



Performanță: alinierea datelor