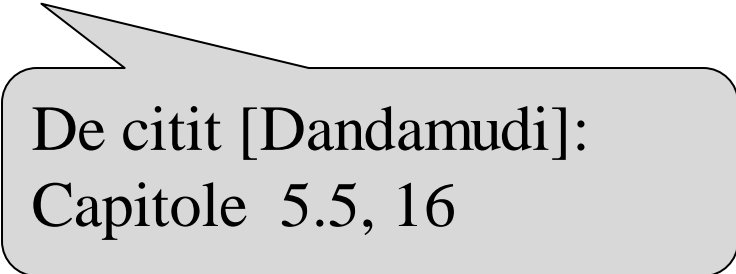


---

# Funcții



De citit [Dandamudi]:  
Capitole 5.5, 16

Modificat: 22-Oct-23

# Proceduri

---

- Doua tipuri
  - \* Apelul-prin-valoare
    - » Primește numai valori
    - » Similar funcțiilor matematice
  - \* Apelul-prin-referință
    - » Primește pointeri
    - » Manipulează direct zona de memorie a parametrilor

# Instrucțiuni de lucru cu Procedurile

---

- x86 dispune de doua instructiuni: **call** si **ret**
- Instrucțiunea **call** este utilizata pentru a apela o procedură, iar formatul acesteia este:

```
call    proc-name
```

```
nexti: ...
```

**proc-name** - numele procedurii (adresa acesteia)

**nexti** - adresa instructiunii urmatoare

- Acțiunile realizate la apelul unei proceduri:

```
push    nexti ; push return address
```

```
jmp     proc-name ; EIP of the procedure
```

# Instrucțiuni de lucru cu Procedurile (cont'd)

---

- Instrucțiunea **ret** este utilizata pentru a transfera controlul către procedura apelanta
- De unde stie procesorul unde sa se intoarca?
  - \* Foloseste adresa de retur salvata pe stiva la executia instructiunii **call**
  - \* Este important ca TOS sa arate către acesta adresa în momentul execuției instrucțiunii **ret**
- Actiunile realizate la executia lui **ret** sunt:

<b>add</b>	<b>ESP, 4</b>	; pop return address
<b>jmp</b>	<b>[ESP-4]</b>	; from the stack

# Instrucțiuni de lucru cu proceduri

---

- Putem specifica un intreg optional instructiunii **ret**

- \* Formatul acesteia este

**ret            optional\_uint**

- \* Exemplu:

**ret 8**

- Acțiunile realizate in acest caz sunt :

```
add          ESP, 4 + optional_uint
jmp          [ESP - 4 - optional_uint ]
```

# Cum este transferat controlul in program?

Offset	machine code	main:
00000002	E816000000	call sum
00000007	9C3	mov EBX,EAX
		; end of main procedure
		sum:
0000001D	55	push EBP
		ret . .
		; end of sum procedure
		avg:
00000028	E8F0FFFFFF	call sum
0000002D	89D8	mov EAX,EBX
		; end of avg procedure

1D-07 = 16

2D-1D=10

# Transmiterea parametrilor

---

- Transmiterea parametrilor este **diferita** fata de limbajele de nivel înalt (C / C++ / Java)
- In limbaj de asamblare
  - » Toți parametrii necesari trebuie dispuși într-o zona de stocare care poate fi accesata mutual de **apelant** și **apelat** (caller vs callee)
  - » Apoi se apeleaza procedura (a.k.a. **call proc\_name**)
- Tipuri zone de stocare
  - » Registre (se utilizeaza registrii de uz general)
  - » Memorie (este folosita stiva)
- Doua metode de transmitere a parametrilor:
  - » Metoda prin registre
  - » Metoda care utilizeaza stiva

# Transmiterea parametrilor prin registre

---

- Procedura **apelantă** plasează toți parametrii necesari în registre de uz general înainte de a realiza apelul propriu-zis prin instrucțiunea **call**
- Exemple:
  - \* **Demo : PROCEX1 .ASM**
    - » apelul-prin-valoare utilizand metoda registrelor
    - » o procedura care realizeaza o suma simpla
  - \* **Demo : PROCEX2 .ASM**
    - » apelul-prin-referenta folosind metoda registrelor
    - » procedura pentru calculul lungimii unei string



# pro și contra pentru metoda registrelor

---

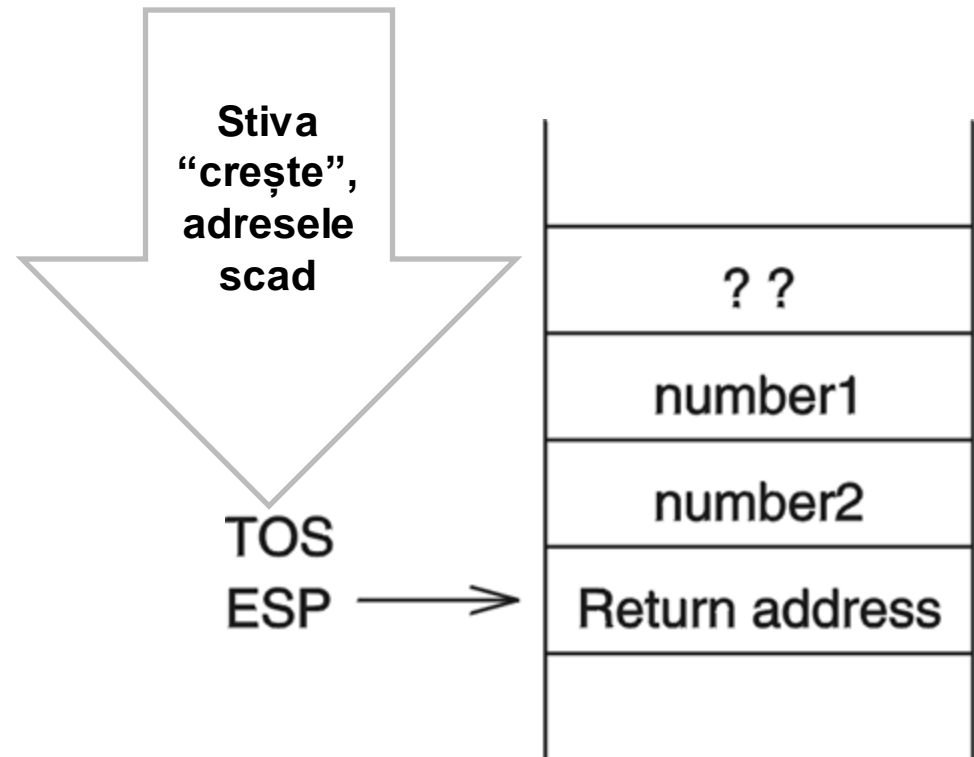
- Avantaje:
  - \* Simplu și convenabil
  - \* Mai rapid
- Dezavantaje:
  - \* Numai un număr limitat de parametri poate fi transferat prin registre
    - Un număr foarte mic de registre este accesibil
  - \* Cel mai adesea registrele nu sunt libere
    - eliberarea acestora prin salvarea lor pe stivă neagă al doilea avantaj al metodei

# Transmiterea parametrilor prin stivă

---

- Toate valorile sunt puse pe stiva înainte de a apel
- Example:

```
push    word number1  
push    word number2  
call    sum
```



# Accesarea parametrilor de pe stivă

---

- Valorile parametrilor se găsesc pe stivă
- Putem folosi următoarea instrucțiune pentru a accesa valoarea parametrului **number2**

```
mov     BX, [ESP+4]
```

## Atenție:

- » ESP se schimbă cu operațiile **push/pop**
- » **A se evita indexarea după ESP**

- Există o alternativă mai bună?
  - \* Folosirea lui EBP pentru a accesa parametrii de pe stivă

# Folosirea lui EBP pentru acces la parametri

---

- Abordarea preferată pentru a accesa parametrii:

```
mov    EBP, ESP
```

```
mov    AX, [EBP+4]
```

pentru a accesa **number2** din exemplul anterior

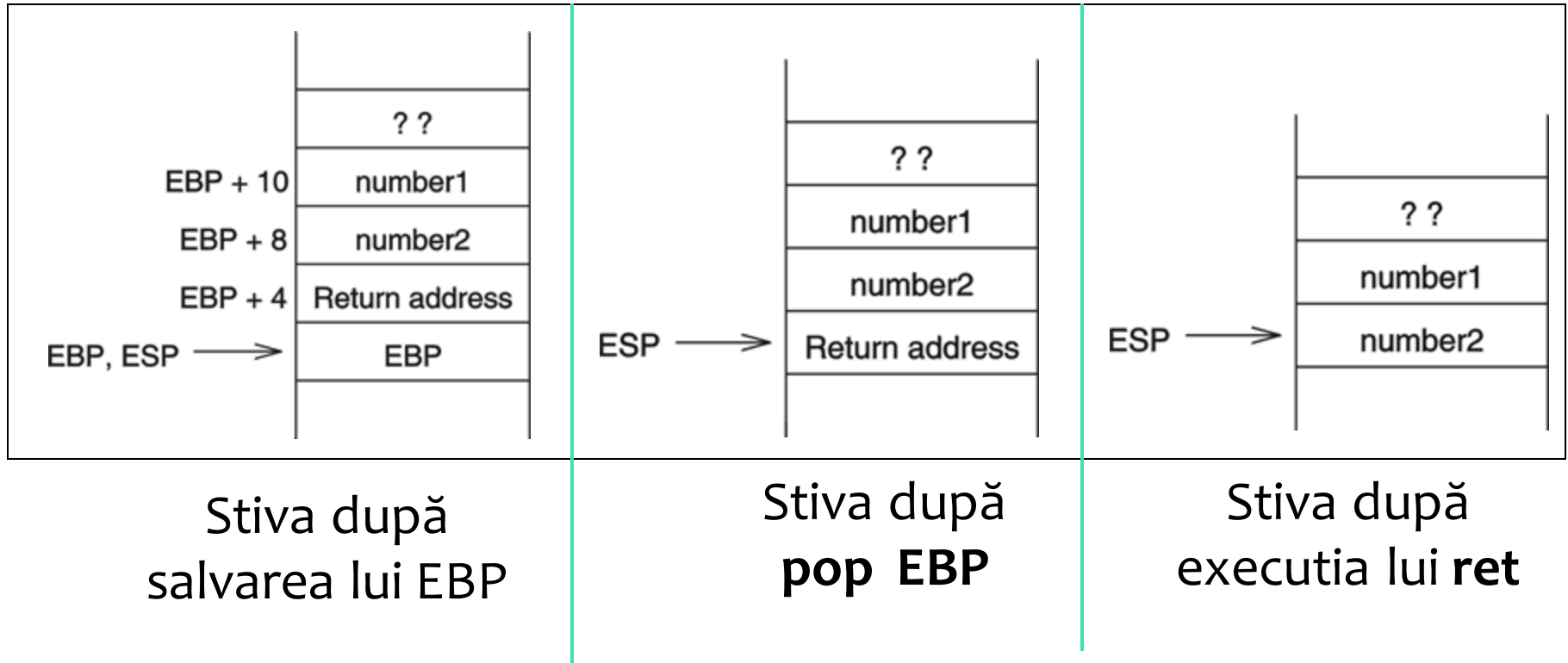
- Problema: Continutul lui EBP este pierdut!

\* Trebuie salvat conținutul lui EBP... pe stivă

```
push   EBP
```

```
mov    EBP, ESP
```

# Eliberarea stivei de parametri



# Eliberarea stivei de parametri (cont'd)

---

- Două moduri pentru eliberarea stivei de parametri:

1. Folosirea intregului optional pentru instructiunea **ret**

**ret        4**

pentru exemplul anterior (2 parametri de 16biți)

2. Adunarea unei constante la ESP în procedura apelantă (C folosește aceasta metodă)

**push        word number1**

**push        word number2**

**call        sum**

**add         ESP, 4**

# Probleme de întreținere a stivei

---

- Cine ar trebui să curețe stiva de parametri?
  - \* **Procedura apelată (callee)**
    - » Codul devine modular (exemplul precedent cu ret 4)
    - » Nu poate fi utilizată cu un număr variabil de parametri
  - \* **Procedura apelantă (caller)**
    - » Trebuie să actualizeze ESP la fiecare apel de procedură
    - » permite număr variabil de parametri ( C )

# Probleme de întreținere a stivei

---

- Trebuie salvat conținutul pentru procedura apelantă
  - » Stiva este utilizata in acest scop
- Care dintre registre trebuie salvate?
  - \* Se salvează acele registre care sunt utilizate de procedura apelantă și sunt modificați de cea apelată
    - » Atenție: setul de registre utilizat variază în timp
  - \* Se salvează toti registrii utilizand **pusha**
    - » Latența crescuta (**pusha** se executa în 5 cicluri de ceas, în timp ce salvarea unui sigur registru se executa într-unul)



# Probleme de întreținere a stivei

---

- Unde se menține starea procedurii apelante?
  - \* Procedura apelantă (caller)
    - » Trebuie cunoscute registrele utilizate de procedura apelata
    - » Trebuie incluse instrucțiuni de salvare și restaurare a registrelor la fiecare apel de procedura
    - » **Cauzează probleme de mentenanță a programului**
  - \* Procedura apelată (callee)
    - » Metoda preferată deoarece codul devine modular (prezervarea stării se realizează într-un singur loc)
    - » Se evită problemele de mentenanță

# Probleme de întreținere a stivei

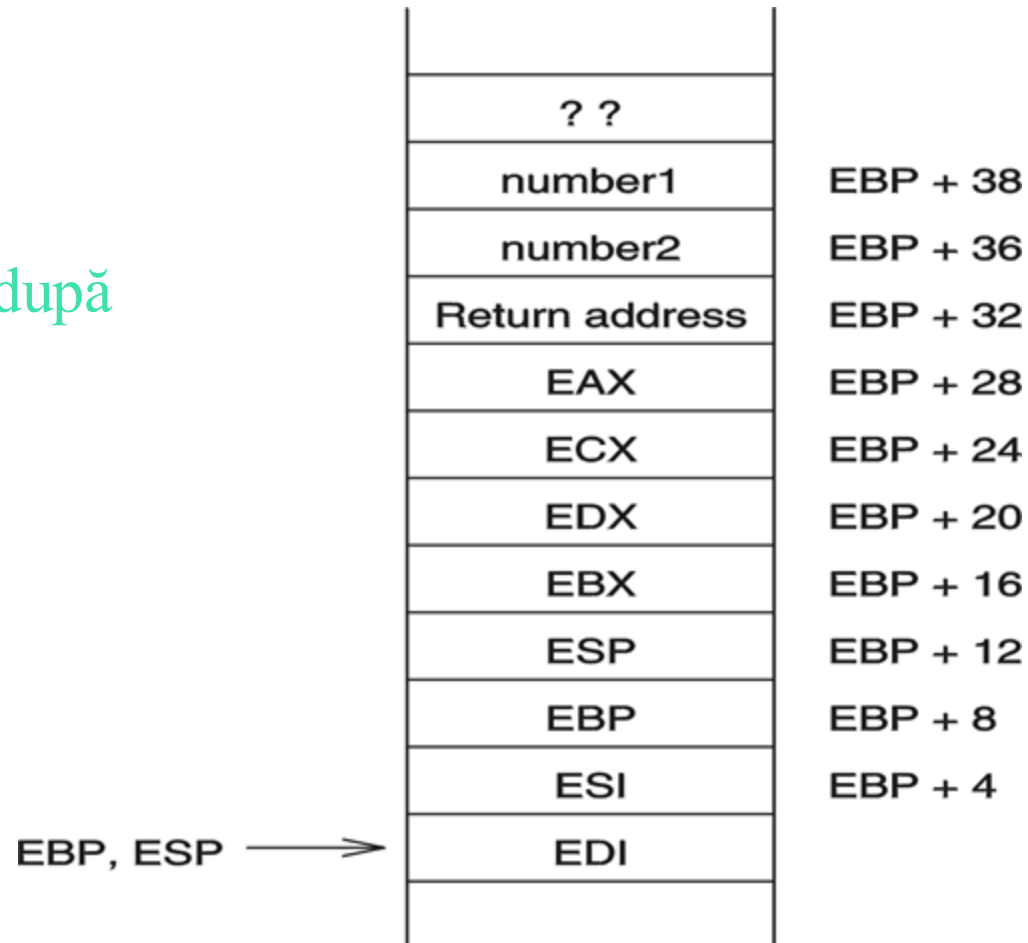
---

- Conservarea stării apelantului în timpul apelului
  - » Pe stivă
- Ce registre ar trebui salvate?
  - \* Se salvează acele registre care sunt utilizate de apelant (caller) și modificate de apelat (callee)
    - » Poate cauza probleme
  - \* Se salvează toate registrele (metoda brute force)
    - » folosind **pusha**
    - » Latență crescută
      - **pusha** se execută în 5 cicluri de ceas, iar salvarea unui registru doar într-unul sigur

# Probleme de întreținere a stivei

---

Starea stivei după  
**pusha**



# Variabile locale

---

- au natură dinamică
  - \* intra în existența la invocarea unei proceduri și se distrug odată cu terminarea execuției acesteia
- nu pot fi în segmentul de date (.data) :
  - » Alocarea spațiului este statică (rămâne persistent între apelurile unei proceduri)
  - » Nu funcționează cu proceduri recursive
- **variabilele locale sunt pe stivă**

# Instrucțiuni pentru cadrul de stivă

---

- Instrucțiunea ENTER

- \* Facilitează alocarea unui cadru de stivă

**enter        bytes , level**

**bytes** = spațiu local de stocare

**level** = nivelul de intercalare (folosim nivelul 0)

- \* Exemplu

**enter        XX , 0**

este echivalent cu

**push        EBP**

**mov         EBP , ESP**

**sub         ESP , XX**

# Instrucțiuni pentru cadrul de stivă

---

- Instrucțiunea LEAVE
  - \* Dealoca un cadru de stiva

**leave**

- » Nu are operanzi
- » Echivalenta cu

**mov           ESP, EBP**

**pop           EBP**

# Schita unei proceduri tipice

---

**proc-name:**

**enter**      **XX**, 0

•   •   •   •   •   •  
**<procedure body>**  
•   •   •   •   •   •

**leave**

**ret**      **YY**

# Transmiterea parametrilor prin stiva - demo

---

- **PROCEX3 .ASM**

- \* apelul prin valoare folosind stiva
- \* o procedura pentru calculul sumei

- **PROCSWAP .ASM**

- \* apelul prin referință folosind stiva
- \* primele doua caractere ale string-ului de input sunt interschimate

- **BBLSORT .ASM**

- \* Implementează algoritmul de sortare prin metoda bulelor
- \* utilizeaza **pusha** si **popa**



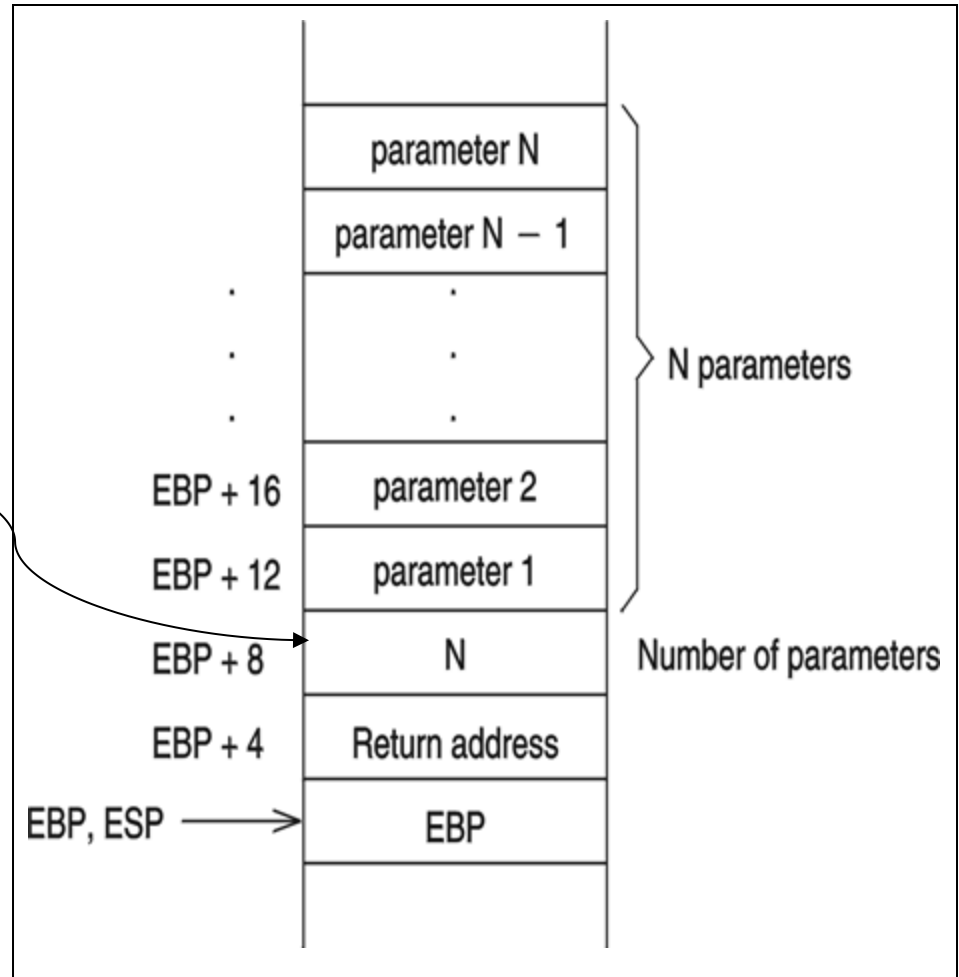
# Număr variabil de parametri

---

- Cele mai multe proceduri au număr fix de parametri
  - \* La fiecare apel același număr de parametri este transmis
- Proceduri cu număr variabil de parametri
  - \* La fiecare apel numărul de parametri poate fi diferit
    - » C folosește acest tip de proceduri
  - \* Ușor de implementat folosind transmiterea prin stivă

# Număr variabil de parametri

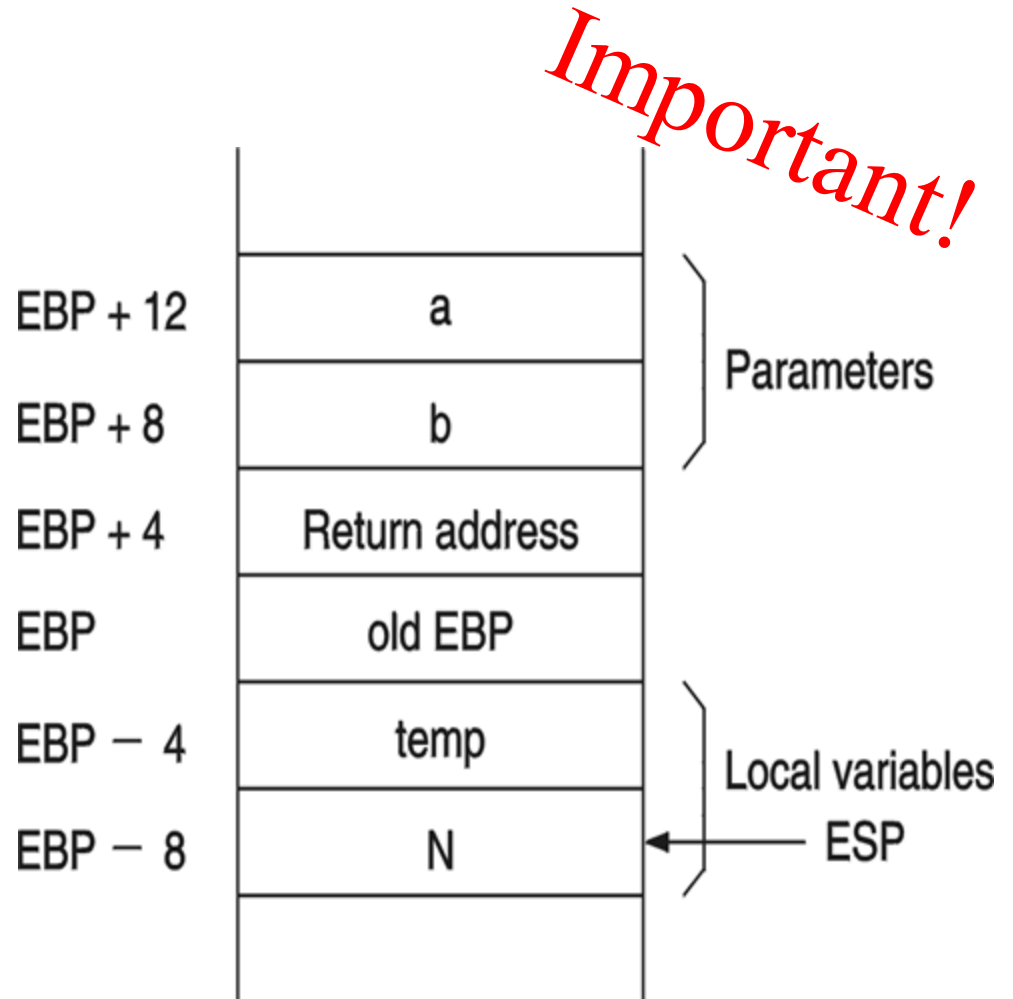
- \* Numărul de parametri trebuie să fie unul din parametri transmiși
- \* Acest număr trebuie să fie ultimul parametru pus pe stivă



# Stack frame = activation record = cadru de stivă

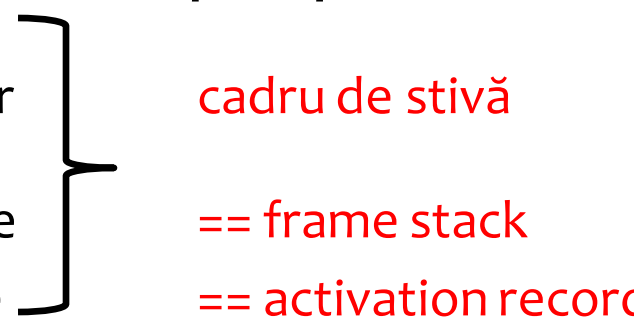
```
void f(int b, int a){  
    int temp=1, N;  
    ...  
    return;  
}
```

```
...  
call    f  
add     ESP, 8  
...  
f:  
    enter 8, 0  
    mov   [EBP-4], 1  
    ...  
    leave  
    ret
```



# Activation record

---

- Datele de pe stivă despre procedura curentă
  - » parametri
  - » adresa de retur
  - » vechiul EBP
  - » registre salvate
  - » variabile locale

The diagram shows a list of five items: » parametri, » adresa de retur, » vechiul EBP, » registre salvate, and » variabile locale. A large right-facing curly bracket groups these items. To the right of the bracket, the text 'cadru de stivă' is aligned with the first three items, '== frame stack' is aligned with the next two, and '== activation record' is aligned with the last one.

  - cadru de stivă
  - == frame stack
  - == activation record
- Fiecare apel de funcție necesită aceste informații
- EBP este denumit **base pointer**
  - \* EBP cunoscut => acces la toate datele din stack frame
  - \* Lista înlănțuită de stack frame-uri
  - \* [EBP] = EBP din funcția apelantă

# Variabile locale – exemple

---

- **curs11/procfib1.asm**

- \* In cazul procedurilor simple, registrele pot fi folosite pentru stocarea variabilelor locale
- \* Utilizarea registrelor pentru stocarea variabilelor locale
- \* Afisarea celui mai mare numar Fibonacci mai mic decat un numar dat ca input

- **curs11/procfib2.asm**

- \* Parametru în EDX, rezultat în EAX
- \* stiva pentru stocarea variabilelor locale
- \* Aspecte legate de performanta utilizarii registrelor vs stiva vor fi discutate ulterior

# Performanță în apeluri de funcții

---

## Stiva versus Registre curs11/bblsort.asm

- *Fara procedura swap (Program 5.5, lines 95-99)*

*swap:*

```
mov    [ESI+4],EAX
mov    [ESI],EBX
mov    EDX,UNSORTED
```

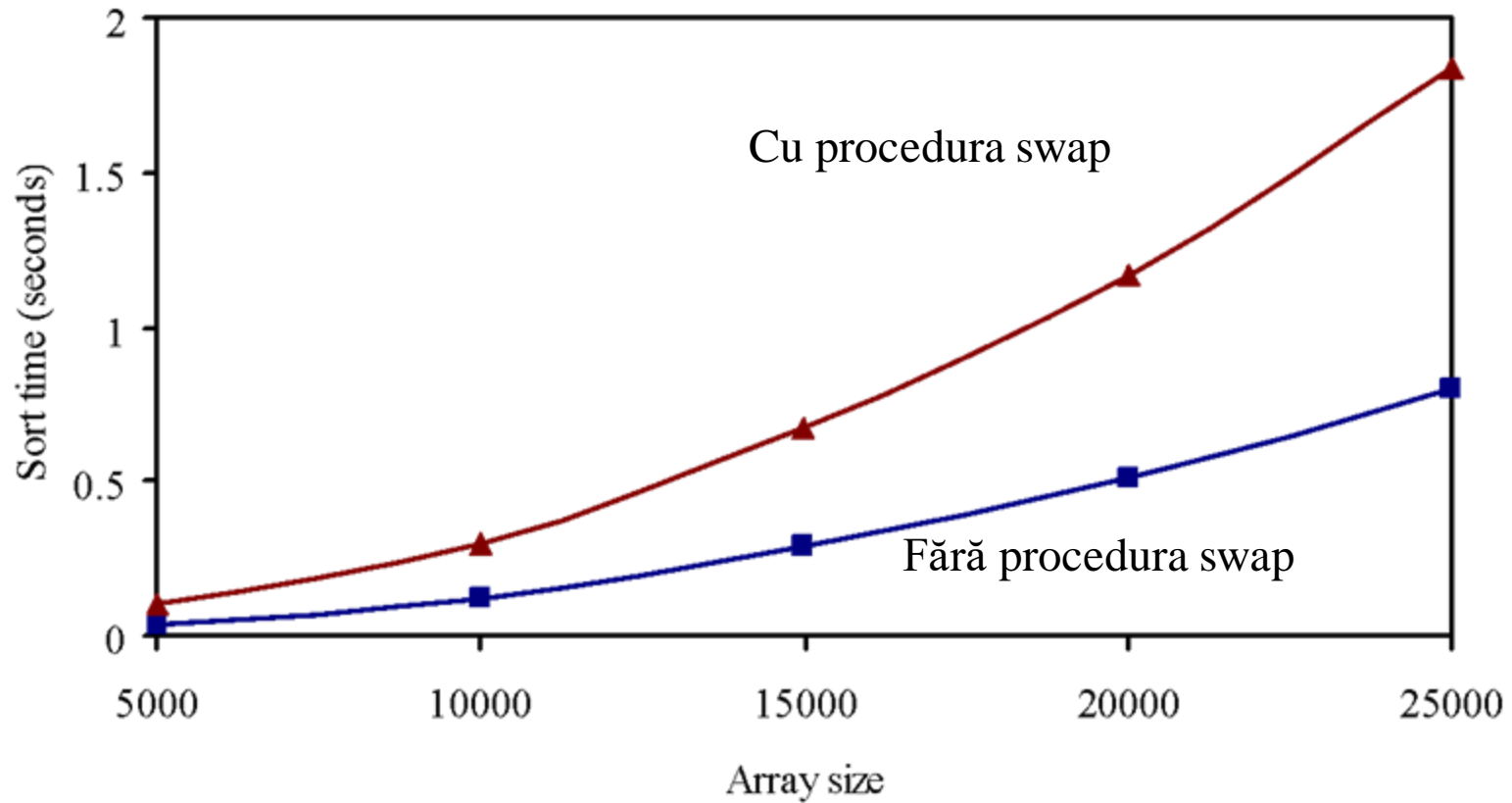
- *Procedura SWAP (inlocuieste codul de mai sus)*

*swap\_proc:*

```
mov    [ESI+4],EAX
mov    [ESI],EBX
mov    EDX,UNSORTED
ret
```

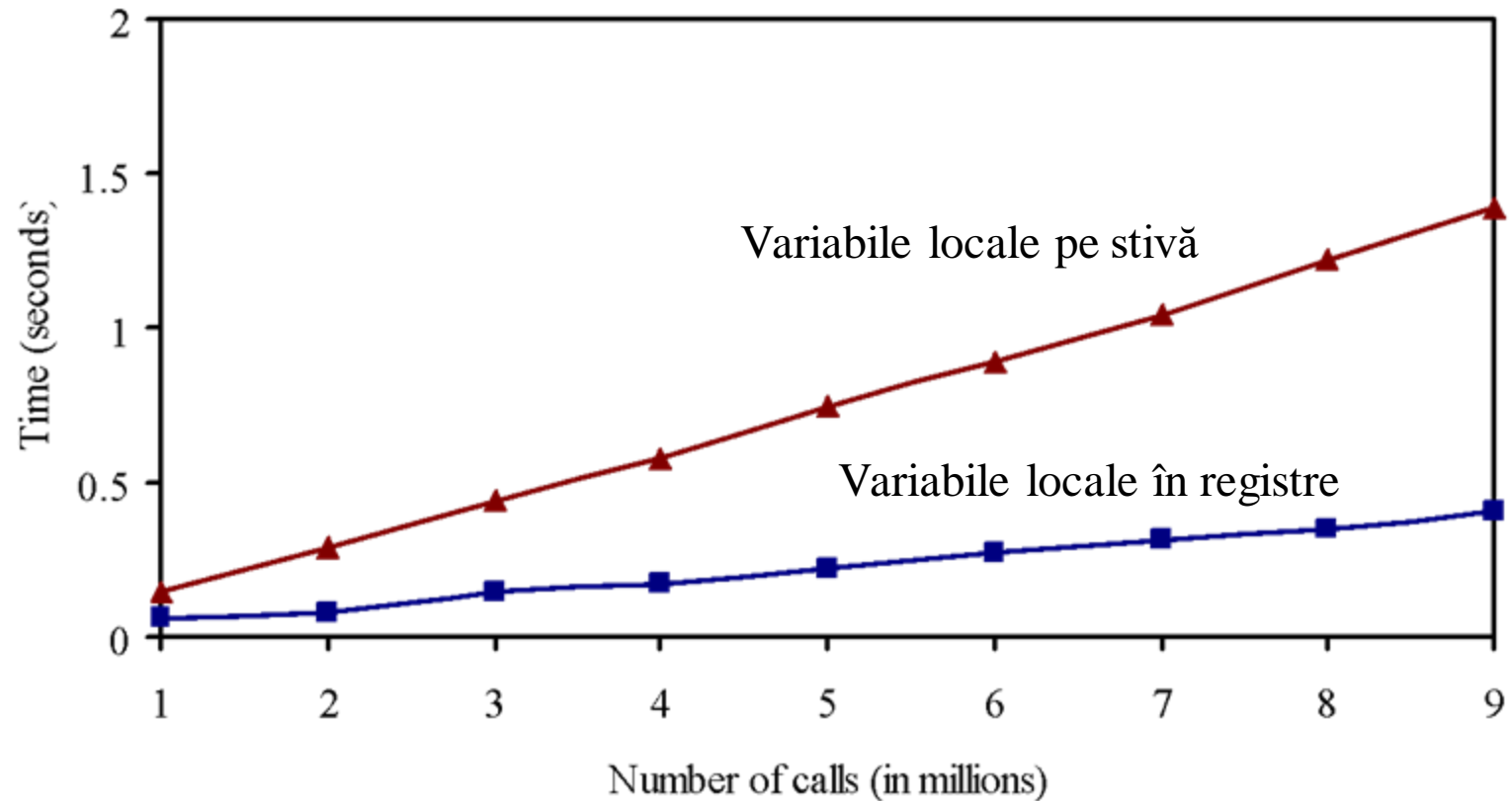
# Performanță în apeluri de funcții

---



# Performanta: Overhead variabile locale

---





---

# Recursivitate

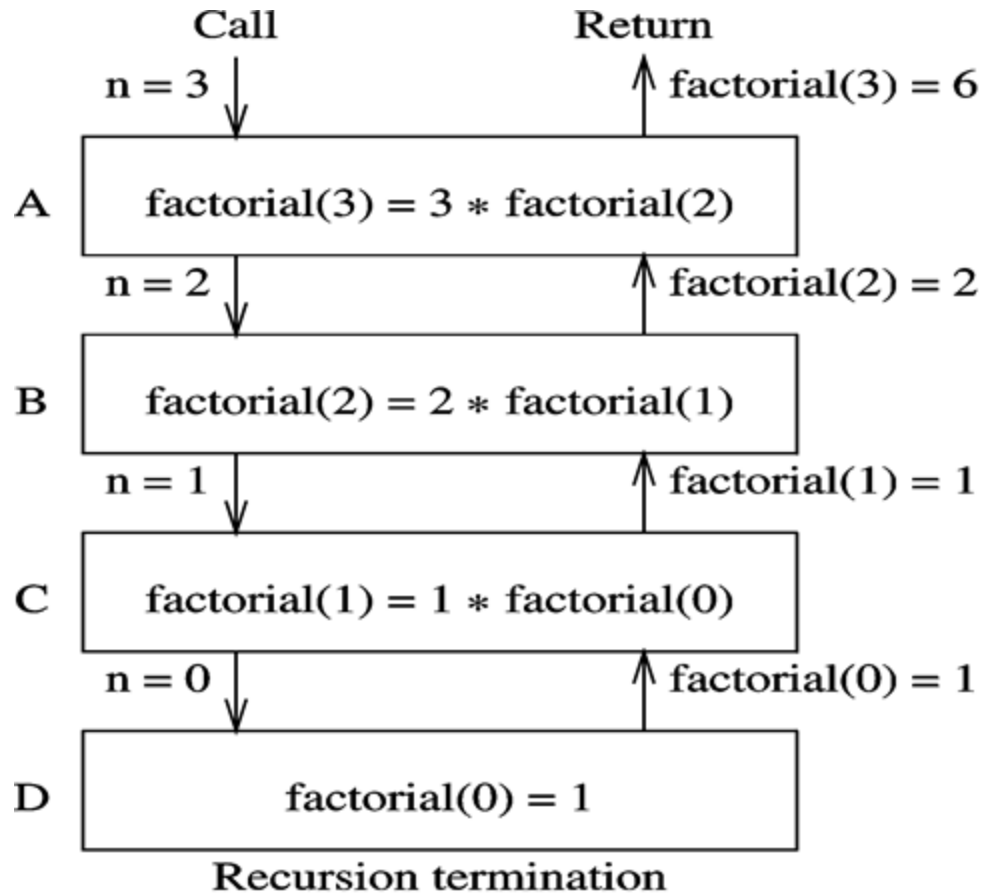
Chapter 16  
S. Dandamudi

# Introducere

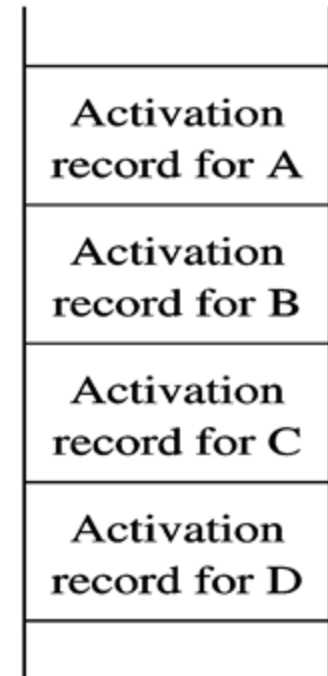
---

- funcție recursivă = care se apelează pe sine însăși
  - \* Direct, sau indirect
- aplicații exprimate natural prin recursivitate
  - $\text{factorial}(0) = 1$
  - $\text{factorial}(n) = n * \text{factorial}(n-1)$  for  $n > 0$
  - Fibonacci, Ackerman, etc
- Din punct de vedere al implementării
  - Similar cu orice alt apel de funcție
  - La fiecare apel se creează un stack frame

# Introdudere (cont'd)



(a)



(b)

# Recursivitate

---

- Doua exemple
  - \* Factorial curs-11/fact\_pentium.asm
  - \* Quicksort curs-11/qsort\_pentium.asm

## Exemplu 1

- \* Factorial
$$\text{factorial}(0) = 1$$
$$\text{factorial}(n) = n * \text{factorial}(n-1) \text{ for } n > 0$$
- **Activation record**
  - \* Consta doar în stocarea adresei de retur pe stivă cu ajutorul instructiunii call
  - \* Parametru în BL

# Recursivitate

---

## Exemplu 2

- Quicksort
  - \* Sortarea unui vector de  $N$  elemente
  - \* Algoritm
    - » Selectam un element pivot  $x$
    - » Presupunem ca ultima aparitie a lui  $x$  este  $\text{array}[i]$
    - » Mutam toate elementele mai mici decat  $x$  pe pozitiile  $\text{array}[0] \dots \text{array}[i-1]$
    - » Mutam toate elementele mai mari decat  $x$  pe pozitiile  $\text{array}[i+1] \dots \text{array}[N-1]$
    - » Aplicam quicksort recursiv pentru a sorta cele doua subliste

# Recursiv vs Iterativ

---

- Recursiv
  - \* Concis
  - \* Mentenanta mai usoara a programului
  - \* Alegerea naturala pentru unele probleme
- Posibile probleme
  - \* Ineficient
    - » Apelurile recursive produc overhead
    - » Calcule duplicate
  - \* Necesita mai multa memorie
    - \* Cadre de stivă

# Recursivitatea la coadă

---

- Tail recursion
- **Numai** când ultima instrucțiune este apelul recursiv
- se poate optimiza apelul recursiv ca un jmp
- nu se mai creează o activare pe stivă
- Exemplu curs-10/tail\_rec
- make && make asm
- Examinare fact.s și fib.s
- <https://gcc.gnu.org/>  
Cu opțiunile -O1 -O2 -m32 -march=native se examinează codul generat pentru recursivitate