

# Laborator 11 - Suport teoretic

---

## Programare multi-modul (asm+asm)

### Programare multi-modul (asm+asm)

Programele non-triviale (de amploare) prezintă problematici specifice complexității crescute a codului, necesitând ca atare și instrumente care adresează aceste aspecte. Natural apar următoarele întrebări:

- cum este posibil a fi împărțită (descompusă) problema atacată în sub-probleme de dificultate redusă?
- care dintre sub-problemele identificate după descompunere sunt deja cunoscute și au rezolvări consacrate, bine cunoscute și care pot fi refolosite?

### Subprograme în limbajul de asamblare x86

- O variantă pentru împărțirea codului în sub-probleme o reprezintă modularizarea codului. Limbajul de asamblare nu recunoaște noțiunea de *subprogram*. Putem însă crea o secvență de instrucțiuni care să poată fi apelată din alte zone ale programului și după terminarea ei să returneze controlul programului apelant.
- O astfel de secvență se numește *procedură*. Apelul unei proceduri se poate face cu o instrucțiune *jmp*. Problema care apare la un astfel de salt este că procesorul nu ține minte de unde a fost trimis la "procedură" și prin urmare nu știe unde să revină cu execuția după terminarea procedurii. Este necesar deci ca la apelul unei proceduri să salvăm undeva adresa de revenire, iar revenirea din procedură este de fapt o instrucțiune de salt la acea adresă.
- Locul unde se salvează adresa de revenire este *stiva de execuție*. Este nevoie de stivă deoarece o procedură poate apela o altă procedură, acea procedură poate apela alta, s.a.m.d.
- Există două instrucțiuni ce permit apelul și revenirea din proceduri: *call* și *ret*.

#### **Sintaxa:**

##### **call eticheta**

- Instrucțiunea *call* este de fapt o instrucțiune *jmp* care în plus introduce în vârful stivei adresa instrucțiunii care urmează - valoarea din EIP (instrucțiunea care vine imediat după *call* și nu destinația saltului produs de instrucțiunea *call*).
- Instrucțiunea *ret* extrage din vârful stivei o adresă și execută un salt la acea adresă (practic se modifică EIP, valoarea extrasă de pe stivă este stocată în registrul EIP). Instrucțiunea nu are argumente deoarece adresa la care sare programul este extrasă din vârful stivei (nu este dată explicit).

### Observație:

- Pentru a evita posibilele ambiguități ce pot să apară atunci când numele unei etichete definite în procedură (etichetă locală) este identic cu numele altei etichete definite în programul "principal", numele etichetelor "locale" trebuie să înceapă cu caracterul "."

### Exemplu

```
.eticheta:; etichetă utilizată în procedură  
eticheta:; etichetă globală
```

- Asamblorul NASM oferă prin intermediul preprocesorului un mecanism simplu prin care un program de multe linii poate fi împărțit în mai multe fișiere. În general, datorită similarității cu directiva *#include* a limbajului C, rolul de utilizare în practică al acesteia este același: permite separarea declarațiilor programului în unul sau mai multe fișiere ce vor fi incluse acolo unde respectivele declarații sunt necesare, întocmai cum și în C se obișnuiește separarea/gruparea declarațiilor în fișiere *header* (fișiere cu extensia .h), fișiere ce sunt ulterior incluse de către codul scris în fișiere C care necesită (referă) declarațiile în cauză.
- La nivel de preprocesor, directiva *%include* îi permite unei componente de program să fie construită din mai multe fișiere ce vor fi asamblate împreună.
- Ca urmare o procedură poate fi definită în același fișier sursă (a se vedea *lab11\_procedura.asm*) sau într-un fișier separat (a se vedea *lab11\_proc\_main.asm* și *factorial.asm*, ambele fișiere sunt în același director, asamblarea, editarea de legături și executia se fac ca și până acum). În aceste cazuri, în urma asamblării veți obține un singur fișier \*.obj!

### lab11\_procedura.asm

```
; programul calculeaza factorialul unui numar si afiseaza in consola  
rezultatul  
; procedura factorial este definita in segmentul de cod si primeste pe stiva  
ca si parametru un numar  
bits 32  
global start  
extern printf, exit  
import printf msvcrt.dll  
import exit msvcrt.dll  
segment data use32 class=data  
    format_string db "factorial=%d", 10, 13, 0  
segment code use32 class=code  
; urmeaza definirea procedurii  
factorial:  
    mov eax, 1  
    mov ecx, [esp + 4]  
    ; mov ecx, [esp + 4] scoate de pe stiva parametrul procedurii  
    ; ATENTIE!!! in capul stivei este adresa de retur  
    ; parametrul procedurii este imediat dupa adresa de retur  
    ; a se vedea desenul de mai jos  
    ;  
    ; stiva  
    ;  
    ;|-----|  
    ;| adresa retur | <- esp  
    ;|-----|  
    ;| 00000006h | <- parametrul pasat procedurii, esp+4
```

```

; |-----|
; ....
.repet:
    mul ecx
    loop .repet ; atentie, cazul ecx = 0 nu e tratat!
    ret
; programul "principal"
start:
    push dword 6          ; se salveaza pe stiva numarul (parametrul
procedurii)
    call factorial        ; apel procedura
    ; afisare rezultat
    push eax
    push format_string
    call [printf]
    add esp, 4*2
    push 0
    call [exit]

```

***lab11\_proc\_main.asm - Diferența față de lab11\_procedura.asm este ca procedura factorial este definită în alt fișier (factorial.asm) fiind necesară includerea acestuia folosind directiva %include.***

```

; programul calculeaza factorialul unui numar si afiseaza in consola
rezultatul
; procedura factorial este definita in fisierul factorial.asm
bits 32
global start
import printf msvcrt.dll
import exit msvcrt.dll
extern printf, exit
; codul definit in factorial.asm va fi copiat aici
#include "factorial.asm"
segment data use32 class=data
    format_string db "factorial=%d", 10, 13, 0
segment code use32 class=code
start:
    push dword 6
    call factorial
    push eax
    push format_string
    call [printf]
    add esp, 2*4
    push 0
    call [exit]

```

### ***factorial.asm***

```

#ifdef _FACTORIAL_ASM_ ; continuiam daca _FACTORIAL_ASM_ este nedefinit
#define _FACTORIAL_ASM_ ; si ne asiguram ca devine definit
; astfel %include permite doar o singura includere
;definire procedura
factorial: ; int _stdcall factorial(int n)
    mov eax, 1
    mov ecx, [esp + 4]
    ; mov ecx, [esp + 4] scoate de pe stiva parametrul procedurii
    ; pentru explicatii a se vedea lab11_procedura.asm
    repet:
        mul ecx

```

```

        loop repet ; atentie, cazul ecx = 0 nu e tratat!
        ret 4
%endif

```

## Programe din mai multe module

Un program scris în limbaj de asamblare poate fi împărțit în mai multe fișiere sursă, fiecare fiind asamblat separat în fișiere *.obj*. Pentru a scrie un program din mai multe fișiere sursă trebuie să respectăm următoarele:

- toate segmentele vor fi declarate cu modificatorul *public*, pentru că în programul final segmentul de cod este construit din concatenarea segmentelor de cod din fiecare modul; la fel și segmentul de date.
- etichetele și numele variabilelor dintr-un modul care trebuie "exportate" în alte module trebuie să facă obiectul unor declarații *global*
- etichetele și variabilele care sunt declarate într-un modul și sunt folosite în alt modul trebuie să fie "importate" prin directiva *extern*
- o variabilă trebuie declarată în întregime într-un modul (nu poate fi jumătate într-un modul și jumătate într-altul). De asemenea, trecerea execuției dintr-un modul în altul se poate face doar prin instrucțiuni de salt (*jmp*, *call* sau *ret*).
- punctul de intrare este prezent doar în modulul ce conține "programul principal"

Fiecare modul se va asambla separat, folosind comanda:

```
nasm -fobj nume_modul.asm
```

apoi modulele se vor lega împreună cu comanda

```
alink modul_1.obj modul_2.obj ... modul_n.obj -oPE -subsys console -entry start
```

## Etape asamblare / link-editare / debugging / execuție

### ASAMBLARE:

```
nasm -f obj modul.asm
```

- Opțiunea -f indică tipul de fișier care să fie generat, în cazul acesta fișier *obj*.

### LINK-EDITARE

```
alink modul1.obj ... moduln.obj -o PE -subsys console -entry start
```

- În folder-ul *nasm* din *asm\_tools* există fișierul "ALINK.TXT" care descrie opțiunile pentru *alink*.

- Alink options:

**-o xxx**

**xxx:**

- COM = output COM file

- EXE = output EXE file
  - PE = output Win32 PE file (.EXE)
- 

#### **-subsys xxx:**

Optiunea specifica tipul de aplicatie generata (default=windows)

---

- windows, win or gui => windows subsystem
  - console, con or char => console subsystem
  - native => native subsystem
  - posix => POSIX subsystem
- 

#### **-entry name**

Optiunea specifica punctul de intrare in program (prima instructiune de executat)

---

### **DEBUG**

`OLLYDBG.EXE modul.exe`

### **EXECUTIE**

`modul.exe`

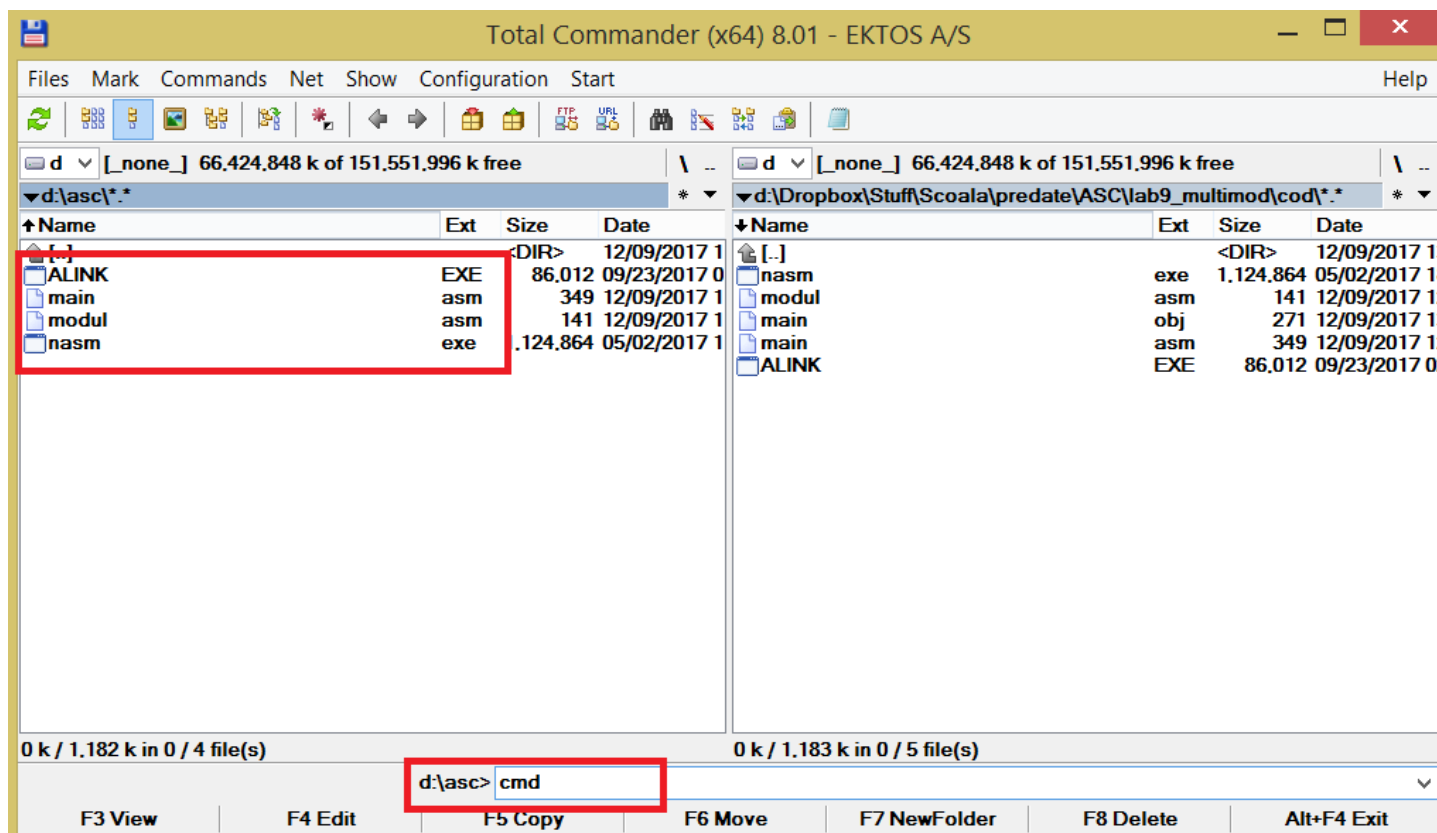
# Laborator 11 - Exemple

## Programare multi-modul (asm+asm)

### Exemple

Se va scrie un program care calculeaza factorialul unui numar. "Programul principal" este *main.asm*, iar funcția factorial este definită în *modul.asm*. Pentru a putea asambla și edita legături este nevoie de linia de comandă. Mai jos este prezentat procesul de asamblare, editare de legături ce duce la un program executabil:

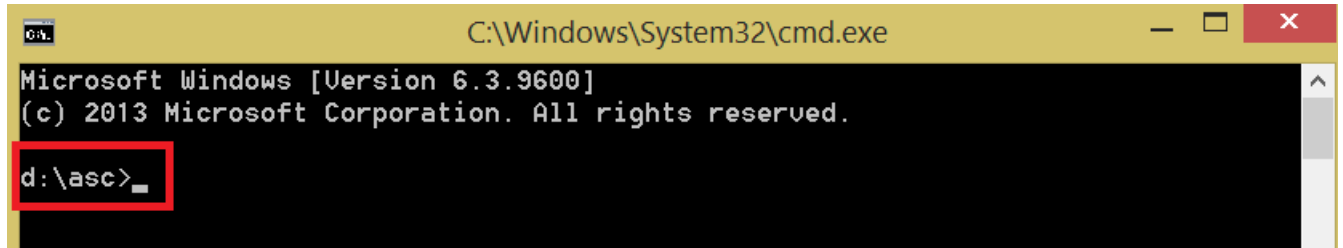
- Fie directorul *asc* pe discul *D*, în acest director se găsesc sursele programului (*main.asm* și *modul.asm*);
- Este necesar ca în directorul *asc* să se copieze și *nasm.exe* și *alink.exe* (programe disponibile în directorul *nasm* din setul de instrumente);
- Pentru a usura navigarea în linia de comandă către locația pe disc unde se află sursele se poate folosi programul Total Commander, se navighează vizual până în directorul *asc* și se lansează linia de comandă (a se vedea figura de mai jos)



- dacă nu se vrea utilizarea programului Total Commander, se lansează *cmd.exe* și cu următoarea secvență de comenzi se poate naviga în directorul *asc*

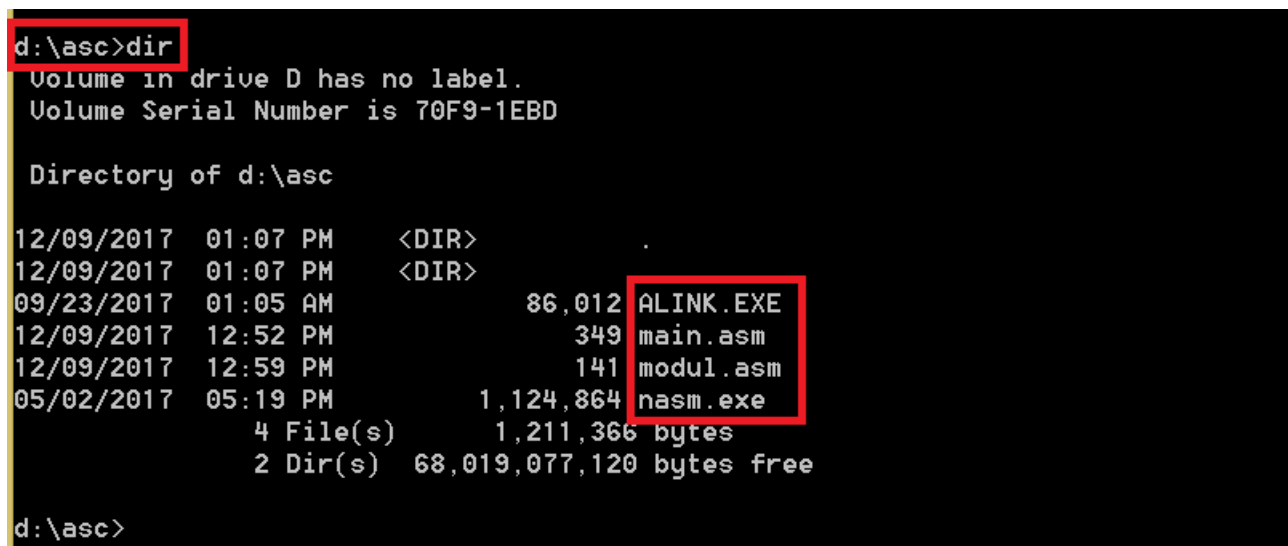
```
> d:
> cd asc
```

- indiferent de varianta aleasă rezultatul trebuie să fie cel din figura de mai jos



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.
d:\asc>
```

- comanda *dir* listează conținutul directorului curent, se verifică dacă directorul în care s-a navigat conține sursele *.asm* și programele *alink.exe* și *nasm.exe* (a se vedea figura de mai jos)



```
d:\asc>dir
Volume in drive D has no label.
Volume Serial Number is 70F9-1EBD

Directory of d:\asc

12/09/2017  01:07 PM    <DIR>          .
12/09/2017  01:07 PM    <DIR>          ..
09/23/2017  01:05 AM      86,012 ALINK.EXE
12/09/2017  12:52 PM       349 main.asm
12/09/2017  12:59 PM       141 modul.asm
05/02/2017  05:19 PM   1,124,864 nasm.exe
               4 File(s)      1,211,366 bytes
               2 Dir(s)  68,019,077,120 bytes free

d:\asc>
```

- se assemblează cele două surse folosind secvența de comenzi

```
> nasm -fobj modul.asm
> nasm -fobj main.asm
```

- rezultatul este cel din figura de mai jos, două fișiere *.obj*

```
d:\asc>nasm -fobj modul.asm
```

1. asamblare modul.asm

```
d:\asc>nasm -fobj main.asm
```

2. asamblare main.asm

```
d:\asc>dir
```

Volume in drive D has no label.

Volume Serial Number is 70F9-1EBD

Directory of d:\asc

```
12/09/2017  01:30 PM    <DIR>          .
12/09/2017  01:30 PM    <DIR>          ..
09/23/2017  01:05 AM             86,012 ALINK.EXE
12/09/2017  12:52 PM             349 main.asm
12/09/2017  01:30 PM             271 main.obj
12/09/2017  12:59 PM             141 modul.asm
12/09/2017  01:30 PM             128 modul.obj
05/02/2017  05:19 PM          1,124,864 nasm.exe
               6 File(s)             1,211,765 bytes
               2 Dir(s)  68,019,077,120 bytes free
```



rezultat pas 2



rezultat pas 1

```
d:\asc>
```

- folosind alink.exe (editor de legături), din cele două fișiere .obj se creează programul executabil, numele fișierului .exe rezultat este identic cu primul fișier .obj dat ca și parametru editorului de legături, în acest caz *main*. Programul poate fi rulat, în acest caz *main.exe* afișează pe ecran  $6! = 720$  (a se vedea figura de mai jos)



```
d:\asc>alink main.obj modul.obj -oPE -subsys console -entry start
```

```
HLINK 01.6 (C) Copyright 1998-9 Anthony H.J. Williams.  
All Rights Reserved
```

```
Loading file main.obj  
Loading file modul.obj  
matched Externs  
matched ComDefs  
Generating PE file main.exe
```

```
d:\asc>dir  
Volume in drive D has no label.  
Volume Serial Number is 70F9-1EBD
```

```
Directory of d:\asc
```

```
12/09/2017  02:10 PM    <DIR>          .  
12/09/2017  02:10 PM    <DIR>          ..  
09/23/2017  01:05 AM             86,012 ALINK.EXE  
12/09/2017  12:52 PM              349 main.asm  
12/09/2017  02:10 PM             2,574 main.exe  
12/09/2017  02:10 PM              271 main.obj  
12/09/2017  01:57 PM              152 modul.asm  
12/09/2017  02:10 PM              124 modul.obj  
05/02/2017  05:19 PM          1,124,864 nasm.exe  
              7 File(s)          1,214,346 bytes  
              2 Dir(s)  68,019,544,064 bytes free
```

```
d:\asc>main  
factorial=720
```

- pentru a depana, codul din ollydbg, File->Open si se deschide fisierul executabil, în acest caz main.exe

## Programe sursa

Exemplul 1:

### 1. lab11\_1.asm

```
; programul calculeaza factorialul unui numar si afiseaza in consola  
rezultatul  
; procedura factorial este definita in segmentul de cod si primeste pe  
stiva ca si parametru un numar  
bits 32  
global start  
  
extern printf, exit  
import printf msvcrt.dll  
import exit msvcrt.dll
```

```

segment data use32 class=data
    format_string db "factorial=%d", 10, 13, 0

segment code use32 class=code
; urmeaza definirea procedurii
factorial:
    mov eax, 1
    mov ecx, [esp + 4]
    ; mov ecx, [esp + 4] scoate de pe stiva parametrul procedurii
    ; ATENTIE!!! in capul stivei este adresa de retur
    ; parametrul procedurii este imediat dupa adresa de retur
    ; a se vedea desenul de mai jos
    ;
    ; stiva
    ;
    ; |-----|
    ; | adresa retur | <- [esp]
    ; |-----|
    ; | 00000006h | <- parametrul pasat procedurii, [esp+4]
    ; |-----|
    ; ....

    .repet:
        mul ecx
    loop .repet ; atentie, cazul ecx = 0 nu e tratat!

    ret

; programul "principal"
start:
    push dword 6          ; se salveaza pe stiva numarul (parametrul
procedurii)
    call factorial        ; apel procedura

    ; afisare rezultat
    push eax
    push format_string
    call [printf]
    add esp, 4*2

    push 0
    call [exit]

```

Exemplul 2:

#### 1. lab11\_proc\_main.asm

---

```

; programul calculeaza factorialul unui numar si afiseaza in consola
rezultatul
; procedura factorial este definita in fisierul factorial.asm
bits 32

global start

```

```

import printf msvcrt.dll
import exit msvcrt.dll
extern printf, exit

; codul definit in factorial.asm va fi copiat aici
#include "factorial.asm"

segment data use32 class=data
    format_string db "factorial=%d", 10, 13, 0

segment code use32 class=code
start:
    push dword 6
    call factorial

    push eax
    push format_string
    call [printf]
    add esp, 2*4

    push 0
    call [exit]

```

## 2. factorial.asm

---

```

#ifdef _FACTORIAL_ASM_ ; continuăm dacă _FACTORIAL_ASM_ este nedefinit
#define _FACTORIAL_ASM_ ; și ne asigurăm că devine definit
                        ; astfel, %include permite doar o singură includere

;definire procedura
factorial: ; int _stdcall factorial(int n)
    mov eax, 1
    mov ecx, [esp + 4]

    repet:
        mul ecx
    loop repet ; atentie, cazul ecx = 0 nu e tratat!

    ret 4 ; in acest caz 4 reprezinta numarul de octeti ce trebuie
    eliberati de pe stiva (parametrul pasat procedurii)

#endif

```

Exemplul 3:

## 1. main.asm

---

```

bits 32
global start

import printf msvcrt.dll
import exit msvcrt.dll
extern printf, exit

```

```

extern factorial

segment data use32
    format_string db "factorial=%d", 10, 13, 0

segment code use32 public code
start:
    push dword 6
    call factorial

    push eax
    push format_string
    call [printf]
    add esp, 2*4

    push 0
    call [exit]

```

## 2. modul.asm

```

bits 32
segment code use32 public code
global factorial

factorial:
    mov eax, 1
    mov ecx, [esp + 4]

    repet:
        mul ecx
    loop repet
    ret 4 ; in acest caz 4 reprezinta numarul de octeti ce trebuie
    eliberati de pe stiva (parametrul pasat procedurii)

```

## Factorial recursiv - Exemplu propus de studentul Molnar Radu, grupa 215

```

bits 32 ; assembling for the 32 bits architecture
; declare the EntryPoint (a label defining the very first instruction of the
program)
global start

; declare external functions needed by our program
extern exit,printf,scanf; adaugam functile externe de care avem nevoie
import exit msvcrt.dll
import printf msvcrt.dll
import scanf msvcrt.dll

; our data is declared here (the variables needed by our program)
segment data use32 class=data
    ; ...
    text db "introduceti un n=",0
    final db "n!=%d",0
    format db "%d",0

```

```

    a resd 1                                ; variabila a va contine numarul n citit de la
tastatura

; our code starts here
segment code use32 class=code
    factor:
        ;pentru a implementa problema recursiv trebuie sa o despartim in
cazuri
        ;la calcularea factorialului recursiv exista doua cazuri:
        ;n!=n*(n-1)!          - iteratia curenta
        ;0!=1                  - conditia de oprire
        ;subprogramul se va reapela pana in momentul in care se ajunge la
ecx=0 moment in care se atribuie eax = 1 si ne intoarcem la pasul anterior
        mov ecx, [esp+4] ;mutam in ecx numarul de pasi pe care ii mai avem de
facut
        jecxz sf ;daca ecx ajunge sa fie 0 sarim la eticheta sf pentru a putea
incepe sa calculam factorialul
        ;daca am trecut de compararea cu 0 atunci ajungem la primul caz al
recursivitatii
        ;formula lui fiind n!=n*(n-1)!
        dec ecx; decrementam ecx pentru a putea sa reapelam functia pentru
pasul urmator
        push ecx; depunem pe stiva valoarea n curenta de calcul a
factorialului
        call factor; apelam functia cu parametrul curent valoarea de pe stiva
        mul dword [esp+8]; inmultim cu valoarea coresp. pasului actual
        add esp,4; eliberam stiva de parametrul utilizat pentru a ajunge la
adresa de revenire a pasului anterior
        jmp return; salt la eticheta return pentru a putea invoca revenirea
din subprogram

    sf:
        mov eax,1;cum recursivitatea noastra are doua cazuri am ajuns in cazul
in care ecx e 0 si returnam 1 - conditia de oprire
        ;0!=1
        return:
        ret ;ne intoarcem la pasul anterior sau in programul principal
start:
    ; ...
    ;tiparirea mesajului
    push dword text
    call [printf]
    add esp,4

    ;citirea lui n de la tastatura
    push dword a
    push dword format
    call [scanf]
    add esp,4*2

    mov ecx,0
    mov eax,0;pregatim registrii pentru apelare
    push dword [a] ;salvam pe stiva numarul n
    call factor ;apelam functia

    ;afisarea rezultatului

```

```
push eax
push final
call [printf]
; exit(0)
push dword 0; push the parameter for exit onto the stack
call [exit]; call exit to terminate the program
```

# Laborator 11 - Probleme propuse

---

## Programare multi-modul (asm+asm)

### Exerciții

Pentru următoarele probleme se cere cel puțin un subprogram implementat într-un modul separat.

1. Se da un număr a reprezentat pe 32 biți fără semn. Se cere să se afișeze reprezentarea în baza 16 a lui a, precum și rezultatul permutărilor circulare ale cifrelor sale.
2. Se cere să se citească de la tastatură un șir de numere, date în baza 10 (se citește de la tastatură un șir de caractere și în memorie trebuie stocat un șir de numere).
3. Se dau două șiruri de caractere. Să se calculeze și să se afișeze rezultatul concatenării celui de-al doilea șir după primul și rezultatul concatenării primului șir după al doilea.
4. Se da un șir de numere. Să se afișeze valorile în baza 16.
5. Se cere să se citească numerele a, b și c și să se calculeze și să se afișeze  $a+b-c$ .
6. Se citesc trei șiruri de caractere. Să se determine și să se afișeze rezultatul concatenării lor.
7. Se dau trei șiruri de caractere. Să se afișeze cel mai lung prefix comun pentru fiecare din cele trei perechi de câte două șiruri ce se pot forma.
8. Să se afișeze, pentru fiecare număr de la 32 la 126, valoarea numărului (în baza 8) și caracterul cu acel cod ASCII.
9. Se cere să se citească de la tastatură un șir de numere, date în baza 16 (se citește de la tastatură un șir de caractere și în memorie trebuie stocat un șir de numere).
10. Se citesc mai multe șiruri de caractere. Să se determine dacă primul apare ca subsecvență în fiecare din celelalte și să se dea un mesaj corespunzător.
11. Se citește de la tastatură un șir de mai multe numere în baza 2. Să se afișeze aceste numere în baza 16.
12. Se dau două șiruri de caractere de lungimi egale. Se cere să se calculeze și să se afișeze rezultatele intercalării literelor, pentru cele două intercalări posibile (literele din primul șir pe poziții pare, și literele din primul șir pe poziții impare).
13. Se dau trei șiruri de caractere. Să se afișeze cel mai lung sufix comun pentru fiecare din cele trei perechi de câte două șiruri ce se pot forma.
14. Se citesc mai multe numere de la tastatură, în baza 2. Să se afișeze aceste numere în baza 8.

### Observatii

- *Se da* înseamnă ca acele date pot fi puse direct în segmentul de date; *se citesc* înseamnă ca acele date trebuie citite de la tastatură.
- Dacă nu este precizat altfel, numerele se considera reprezentate pe 32 biți fără semn, iar șirurile de caractere de până la 100 de caractere (șirul propriu-zis).