



UNIUNEA EUROPEANĂ



GUVERNUL ROMÂNIEI



Instrumente Structurale  
2007-2013



# Platformă de e-learning și curriculum e-content pentru învățământul superior tehnic

## Programare în limbaj de asamblare

**25. Operare pe șiruri, instrucțiuni de transfer al controlului programului, de control procesor și de sistem.**

## Instrucțiuni de operare pe șiruri

Există cinci operații de bază ce operează pe șiruri de lungime de până la un segment (64K la 286, respectiv 4G la 386/486), și permit: mutare, comparare de șiruri, scanarea unui șir pentru o valoare și transfer la/de la acumulator. Aceste operații de bază pot fi precedate de un prefix special de un octet care are ca efect repetarea instrucțiunii prin hardware. Repetiția se poate termina printr-o varietate de condiții, iar o operație repetată poate fi întreruptă sau suspendată.

O instrucțiune pe șir poate avea un operand sursă, un operand destinație sau pe amândoi. Șirul sursă este în (**DS**) sau într-un alt segment, dacă este prefixat peste această presupunere.

Șirul destinație se află, întotdeauna, în segmentul dat de registrul (**ES**).

**SI** este ca offset pentru elementului curent al șirului sursă, iar **DI** este offset pentru șirul destinație. Instrucțiunile pe șir actualizează automat **SI** și/sau **DI**. **DF** determină dacă registrele index sunt auto-incrementate-decrementate ( $DF=0-1$ ), iar pas de actualizare depinde de tipul șirurilor: 1 - octet, 2 - cuvânt, 4 - dublu cuvânt.

Dacă este prezent un prefix de repetare reg. **CX** este decrementat la fiecare repetiție a instrucțiunii pe șir; repetarea va lua sfârșit când  $CX=0$ , sau pentru unele prefixe și în funcție de **ZF**.

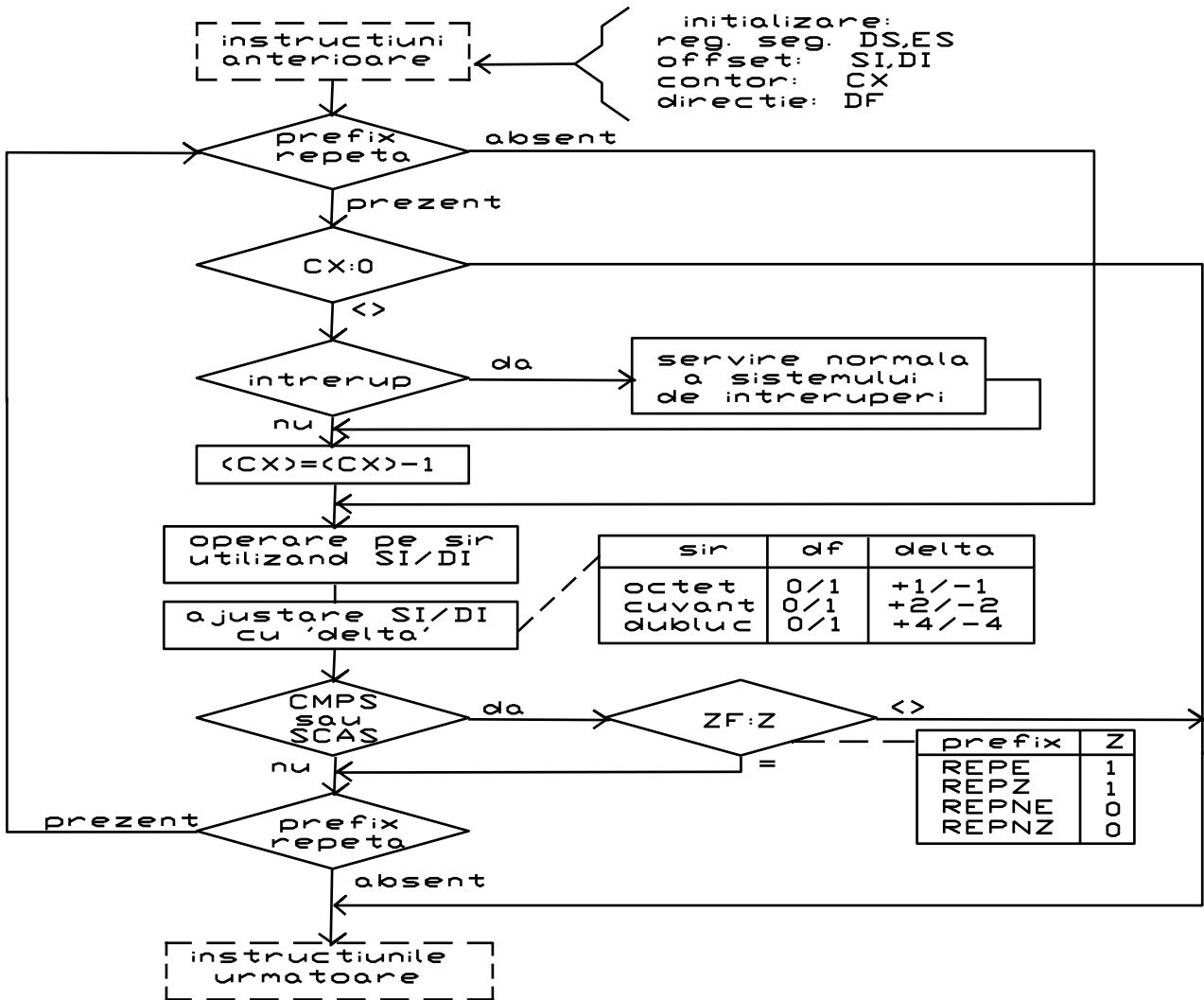
Sunt cinci mnemonici pentru două forme de prefix octet, care controlează repetarea unei instrucțiuni pe șir(uri). Prefixele de repetare nu afectează indicatorii.

### **REP / REPE / REPZ / REPNE / REPNZ**

**REP** - REPeat - este utilizat împreună cu **MOVS**- **STOS**, "repetă cât timp nu este sfârșitul șirului, adică ( $CX \neq 0$ )";

**REPE / REPZ** - REPeat while Equal / Zero - care operează la fel și au fizic același prefix octet ca **REP**. Sunt utilizate pentru **CMPS** și **SCAS**, și necesită, în plus față de condiția anterioară, ca **ZF** să fie setat înainte de efectuarea următoarei repetiții;

**REPNE / REPNZ** - REPeat while Not Equal / Zero - este asemănător cu cele anterioare, cu deosebirea că **ZF** trebuie pus pe 0, sau repetiția se termină.



**MOVS <sir\_dest>, <sir\_sursa>** (MOVE data from String to string)

**MOVSB / MOVSW / MOVSD**

Se transferă un element (octet/cuvânt/dubluc) de la sursă (SI) la destinație (DI), și actualizează SI/DI

Utilizată împreună cu prefixul REP realizează un transfer de bloc memorie-memorie.

**CMPS <sir\_dest>, <sir\_sursa>** (COMPARE String operands)

**CMPSB / CMPSW / CMPSD**

Această instrucțiune scade operandul destinație (referit de DI) din cel sursă (referit de SI), modifică indicatorii de stare, dar nu alterează nici unul dintre operanzi, după care actualizează SI/DI.

Trebuie reținut că spre deosebire de instr. CMP care scade sursa din destinație, CMPS realizează scăderea invers: sursă - destinație.

Dacă CMPS are prefixul REPE sau REPZ operația este interpretată "compară cât timp nu este sfârșit șirul (CX > 0) și șirurile sunt egale (ZF=1)"; determina prima pereche de elemente diferite.

Dacă CMPS este precedată de REPNE/REPZ "compară cât timp nu este sfârșit de șir (CX<>0), și elementele șirurilor nu sunt egale (ZF=0)"; determina prima pereche de elemente egale.

**SCAS <șir\_dest>** (SCAn String data)

**SCASB / SCASW / SCASD**

Instrucțiunea scade elem. șir dest. (octet/ cuvânt/ dublu cuvânt) adresat de DI, din AL, AX sau EAX, și actualizează indicatorii, dar fără a modifica șirul destinație sau acumulatorul.

Dacă SCAS este prefixată de REPE/REPZ "scanează cât timp nu este sfârșitul șirului (CX<>0) și valoarea scanată este egală cu elementele șirului (ZF=1)"; determina abaterea față de o valoare.

Dacă SCAS este prefixată de REPNE/REPZ "scanează cât timp nu este sfârșitul șirului (CX<>0) și valoarea scanată este diferită de elementele șirului (ZF=0)"; localizează o valoare într-un șir.

**LODS <șir\_sursa>** (LOaD String operand)

**LODSB / LODSW / LODSD**

Transferă elementul șirului sursă adresat de SI, în AL, AX sau EAX, și actualizează SI.

Instrucțiunea se utilizează în bucle soft.

**STOS <șir\_dest>** (STOre String data)

**STOSB / STOSW / STOSD**

Transferă un element din acumulator (AL, AX, EAX) în șirul destinație, și actualizează DI, pentru a referi următorul element.

Instrucțiunea poate fi precedată de prefixul REP, și astfel se poate inițializa un șir cu o constantă.

1) Copierea unui șir de octeți dintr-o zonă de memorie într-alta.

```
date_sir    segment    word public 'data'
            sir1    db    1000 dup (7,0f0h)    ; șirul sursă
            lung1   equ    $ - sir            ; lungimea șirului sursă
            sir2    db    1000 dup (2 dup (?)) ; rezervare pt. șir dest.
            ptr_sir1 dd    sir1                ; pointer sir1
            ptr_sir2 dd    sir2                ; pointer sir2
date_sir    ends
cod         segment    word public 'code'
            assume cs:cod, ds:date_sir, es:date_sir
start: mov  ax, date_sir ; inițializare registru segment DS
        mov  ds, ax      ; și apoi adresele celor două șiruri
        mov  es, ax      ; sau: les  di,ptr_sir2
        lea  di, sir2     ;      lds  si,ptr_sir1
        lea  si, sir1
        mov  cx, lung1   ; contorul transferului = lungimea sursei
        cld              ; direcția de parcurgere a șirului (df=0)
repetă: lodsb
        stosb            ; sau: movs sir2,sir1 , sau movsb
        loop repetă      ; sau: rep  movsb
        mov  ax, 4c00h   ; revenire DOS
```

```

        int    21h
cod     ends
end     start

```

2) Copierea informației dintr-un buffer din memorie în memoria ecran. În modul text pentru fiecare caracter de pe ecran se rezervă doi octeți: unul reprezintă atributele de afișare (afișare continuă / intermitentă - 1 bit, culoare fond - 3 biți, culoare caracter - 4 biți), iar cel de-al doilea este codul ASCII al caracterului afișat.

```

.model small
.data
mem_ecran  dd    0B800000h ; adresa memoriei ecran
dimensiune equ  2000
buffer dw dimensiune dup (3a33h); se va afișa același caracter
ptr_buf    dd    buffer
.code
start: mov  ax, @data    ; inițializare DS
        mov  ds, ax
        mov  ah, 0      ; selecție mod de lucru
        mov  al, 0      ; alfanumeric 40*25 alb/negru, pt. 80*25 al=2
        int  10h
        les  di, mem_ecran
        lds  si, ptr_buf
        mov  cx, dimensiune
        cld                      ; direcția de parcurgere
        rep movsw                 ; transfer buffer în mem_ecran
        mov  ax, 4c00h           ; revenire DOS
        int  21h
.stack 10h
end     start

```

3) Determinarea poziției unui anumit caracter (sau a unui șir de caractere) într-un fișier sursă de pe disc.

```

.model small
.stack 10h
.data
car equ 'A' ; caracterul de identificat (sau șirul)
lung equ 2048; dimensiunea maximă a fișierului-4 sectoare
buffer db lung dup (?); spațiu de mem. pentru fișier
nume_fis db 'fișier.asm', 0 ; nume fiș.- ASCIIZ, adică după
        ; numele poate fi precedat și de calea de acces
pozitie dw ? ; poziția caracterului în fișier
nr_logic dw ? ; numărul logic atribuit fișierului
contor dw ? ; numărul efectiv de caractere citite
mes_lipsa db 'nu exista fișier cu acest nume$'
mes_err_cit db 'eroare de citire de pe disc$'
mes_car_lipsa db 'caracterul cautat nu este in fișier$'

```

```

.code
start: mov ax, @data ; inițializare DS
      mov ds, ax
      mov ah, 3dh ; apel DOS pentru deschidere de fișier
      mov al, 0 ; în modul "citește" (1-scrie,2-citire/scriere)
      lea dx, nume_fis ; adresa numelui fișierului
      int 21h ; apel funcție 'deschide fișier'
      jc lipsa_fis
      mov nr_logic, ax ; se depune numărul sectorului
      mov bx, ax ; și în BX, pentru funcția de citire
      mov cx, lung ; contor număr maxim de caractere citite
      lea dx, buffer ; adresa unde se vor depune caracterele
      mov ah, 3fh ; funcția de citire din fișier
      int 21h
      jc err_cit ; dacă a apărut eroare al citire
      mov contor, ax ; la 'contor' se depune numărul de car.
      mov cx, ax ; contor de căutare 'car'
      push ds ; se încarcă în ES adresa de segment
      pop es ; a 'buffer'-ului
      lea di, buffer ; și în registrul (DI) offsetul acestuia
      mov al, car ; caracterul de căutat
      cld ; stabilire direcție de parcurgere, (DF)=0
repne scasb ; continuă scanarea până-l găsește
      je gasit ; dacă nu s-a găsit nu se face saltul
      lea dx, mes_car_lipsa ; și se tipărește mesajul
      mov ah, 9 ; mesajul: 'caracterul cautat nu este în fișier'
      int 21h
      jmp gata
gasit: dec di ; poziția caracterului căutat, (DI)-1
      mov pozitie, di ; deoarece (DI) a fost actualizat după jmp gata
      ; scanare
lipsa_fis:
lea dx, mes_lipsa ; nu s-a găsit fișierul cu
mov ah, 9 ; numele specificat
int 21h
jmp gata
err_cit:
lea dx, mes_err_cit ; eroare la citirea fișierului
mov ah, 9
int 21h
gata: mov ah, 3eh ; închiderea fișierului deschis
mov bx, nr_logic
int 21h
mov ax, 4c00h ; revenire DOS
int 21h
end start

```

Determinarea poziției unui șir de caractere într-un fișier:

```
    sir_cardb    'asamblare'
    lung_sir    dw    $-sir_car
    ptr_sir     dd    sir_car
    ptr_buf     dd    buffer
    pozitie     dw    ?      ; poziția șirului de caractere
    .....
    les    di, ptr_buf
    lds    si, ptr_sir
    mov    cx, contor    ; dimensiune buffer
    sub    cx, lung_sir  ; căutarea se va face începând de la 0
    inc    cx            ; până la (contor) - (lung_sir)
    cld                    ; direcția de parcurgere (DF=0)
reia:  push si; se salvează adresa șirului căutat
    push di    ; se salvează adresa șirului în care se caută
    push cx; se salvează contorul numărului maxim de căutări
    mov    cx, lung_sir  ; contorul de comparații
    repe cmps buffer, sir_car    ; se compară cât timp sunt egale
    pop    cx    ; reface resurse salvate în stivă: contorul
    pop    di    ; și adrese șiruri: destinație - șirul în care se
    pop    si    ; caută, sursă – (sub)șirul care se caută
    je    gasit   ; salt dacă s-a găsit sir_car în buffer
    inc    di    ; căutarea se va relua de la următorul caracter
    loop  reia   ; din buffer, dacă (contor) <> 0
; -> șirul de caractere nu este în fișierul dat (respectiv în buffer)
    jmp    nu_gasit
gasit: mov    pozitie, di    ; valoarea salvată pentru (DI)
    .....
nu_gasit:    ; tipărire mesaj: 'Nu s-a gasit șirul de caractere'
```

## Instrucțiuni de transfer al controlului programului

Instrucțiunile de transfer al controlului programului operează asupra lui IP și CS. La apariția unui astfel de transfer, coada de instrucțiuni nu mai conține instrucțiunea următoare și EU va transmite noua adresă la BU, care va obține instrucțiunea următoare direct din memorie, utilizând noile valori pentru CS:IP; instrucțiunea va fi transferată către IU și EU, pentru execuție, și BU apoi începe reumplerea cozii de la noua locație.

Există patru tipuri de instrucțiuni de transfer control:

- instrucțiuni de transfer necondiționat;
- instrucțiuni de transfer condiționat;
- instrucțiuni de ciclare;
- instrucțiuni de întrerupere;

Dintre acestea numai instrucțiunile de întrerupere afectează indicatorii.

## Instrucțiuni de transfer control, necondiționat

Transferul controlului se poate face la o locație (țintă) în segmentul de cod curent (transfer intrasegment/NEAR) sau la un segment de cod diferit (transfer intersegment/FAR).

### CALL <nume\_procedură>

Această instrucțiune activează o procedură, salvând adresa de revenire, în stivă, pentru a permite unei instrucțiuni RET (return), din procedură, să transfere controlul înapoi la instrucțiunea următoare lui CALL.

Asamblorul generează tipuri diferite de instrucțiuni mașină pentru CALL, după modul cum programatorul a definit numele procedurii, cu atributul NEAR sau FAR.

Pentru returnarea corectă a controlului, tipul instrucțiunii CALL trebuie să fie potrivit cu tipul instrucțiunii RET. Diferitele forme ale instrucțiunii CALL permit ca adresa procedurii țintă să fie obținută direct din instrucțiune (direct), sau de la o locație de memorie sau dintr-un registru referit de instrucțiune (indirect).

a) - CALL direct intrasegment:

```
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (IP)
(IP) ← (IP) + deplasament
```

La 386/486 deplasamentul poate avea și 32 biți, deci adunarea se va face modulo 4G.

```
call near_etich
call near_proc
```

b) - CALL direct intersegment:

```
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (CS)
(CS) ← al 2-lea cuvânt din instr. (adresa de segment)
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (IP)
(IP) ← primul cuvânt din instr. (adresa de offset)
```

```
call far_etich;          call far_proc
```

c) - CALL indirect intrasegment:

```
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (IP)
(IP) ← registru / memorie;
```

Exemple:

```
call cx;          call word ptr nume_var
call word ptr [bx][si];  call mem_word
```

d) - CALL indirect intersegment, poate fi făcut numai prin memorie:

```
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (CS)
(CS) ← al 2-lea cuvânt mem. referit de instrucțiune
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (IP)
(IP) ← primul cuvânt mem. referit de instrucțiune
```

```
call dword ptr [bx];          call dword ptr nume_var[bp][si]
```



Când se utilizează apeluri indirecte trebuie asigurat că tipul lui CALL este potrivit cu tipul instrucțiunii de revenire RET->erori.

**RET [val\_pop]** (RETurn from procedure)

Asamblorul generează 2 tipuri de instr. mașină: RET intrasegment dacă procedura a fost definită NEAR, și RET intersegment, dacă procedura este de tip FAR. Valoarea *val\_pop*, care este opțională, este prevăzută pentru a descărca parametrii din stivă.

```
(IP) ← ((SP)+1:(SP))
(SP) ← (SP) + 2
if inter_segment then
    (CS) ← ((SP)+1:(SP))
    (SP) ← (SP) + 2
if val_pop <> 0 then
    (SP) ← (SP) + val_pop
```

Tipul instrucțiunii RET trebuie să se potrivească cu tipul lui CALL:

**call word ptr [bx]** ; pentru o procedură de tip FAR.

- se va salva în stivă doar offsetul de revenire (IP), iar la execuția instrucțiunii RET, din procedura de tip FAR, se va reface din stivă, pe lângă IP și CS, cu următorul cuvânt din stivă (și se va pierde controlul asupra calculatorului).

**call dword ptr [bp][si]** ; pentru o procedură de tip NEAR.

- se vor salva în stivă adresele de revenire pentru CS:IP, dar la terminarea procedurii se va reface din stivă numai IP, rămânând în stivă un cuvânt, care în final va duce la pierderea controlului.

**JMP <țintă>**

Spre deosebire de CALL, JMP nu salvează în stiva nici o informație.

a) *JMP direct intrasegment*, modifică IP prin adunarea deplasamentului relativ al țintei, conținut în instrucțiune la valoarea, actualizată, a lui IP. Auto-relative (relocabile dinamic).

```
jmp near_etich ; salt direct intrasegment
jmp short near_etich; salt de tip short
```

b) - *JMP direct intersegment* (instr. are 5 octeți/286, sau 5/7 la 386):

```
jmp far_etich ; salt în alt segment
jmp far ptr etich ; 'etich' poate fi și în același segment
```

c) - *JMP indirect intrasegment*:

```
jmp ax jmp tabela[bx]
jmp si jmp word ptr [bp][di]
```

d) - *JMP indirect intersegment*, poate fi făcut numai prin memorie: jmp etich\_dword

```
jmp dword ptr [bx][si]
```

Programul următor execută diferite secvențe de program, în funcție de opțiunea utilizatorului, introdusă de la tastatură.

**.model small**

**.data**

```
executie db 0Dh, 0Ah, 'Executa secventa (1,2,3 sau 4=stop):$'
mes_secv1 db 'S-a executat secventa 1',0dh,0ah,$'
mes_secv2 db 'S-a executat secventa 2',0dh,0ah,$'
mes_secv3 db 'S-a executat secventa 3',0dh,0ah,$'
```

```

tab_secv    label  word
            dw    secv1
            dw    secv2
            dw    secv3
            dw    gata

.code
start: mov  ax, @data    ; inițializare adresa de segment
      mov  ds, ax
iar:   lea  dx, executie ; solicită secvența dorită, utilizatorului
      mov  ah, 9         ; se tipărește mesajul de selecție secvență
      int  21h          ; apel funcția 9, tipărire mesaj
      mov  ah, 1         ; apel funcția 1, citire caracter
      int  21h          ; caracter returnat în AL
      sub  al, 31h       ; intervalul '1'÷'4' -> 0÷3 și verifică dacă
      jc   iar           ; este în intervalul 0÷3: dacă nu, cere
      cmp  al, 4         ; o valoare, în acest interval, fără a executa
      jnc  iar           ; vreuna dintre secvențe
      cbw                ; extensie pe 16 biți
      mov  bx, ax
      shl  bx, 1         ; *2, pentru a obține adresa relativă
      jmp  word ptr tab_secv[bx] ; în tabela cu adr. secvențe
secv1: lea  dx, mes_secv1; se execută prima secvență
      mov  ah, 9
      int  21h
      jmp  short iar
secv2: lea  dx, mes_secv2; se execută a doua secvență
      mov  ah, 9
      int  21h
      jmp  short iar
secv3: lea  dx, mes_secv3; se execută a treia secvență
      mov  ah, 9
      int  21h
      jmp  short iar
gata:  mov  ax, 4c00h    ; revenire DOS
      int  21h
.stack 20h
end

```

## Instrucțiuni de transfer control, condiționat

### Jcond <țintă>

Toate salturile condiționate sunt de tip SHORT, adică ținta trebuie să fie în segmentul de cod curent, în intervalul de [-128, +127] octeți față de instrucțiunea de transfer (de ex. JMP 00, realizează saltul la primul octet al următoarei instrucțiuni). La 386/486 aceste instrucțiuni permit și realizarea unui salt de tip NEAR, adică în cadrul aceluiași segment (64K sau 4G, depinde de

atributul de dimensiune adresă). Deoarece adresa de salt este determinată prin adunarea deplasamentului relativ al țintei, la IP-ul actualizat, toate salturile condiționate sunt autorelative. În cazul când se dorește un salt condiționat pe o distanță mai mare de 128 octeți, la 286, sau un salt într-un alt segment, atunci trebuie să se utilizeze două instrucțiuni de salt: un salt condiționat (pentru condiția negată, adică, de exemplu JE, pentru JNE), în intervalul [-128,+127], iar, imediat după instrucțiunea de salt condiționat, o instrucțiune de salt, necondiționat, pe distanța dorită.

**JZ / JE** ZF = 1, dacă este egalitate (d) = (s)  
**JNZ / JNE** ZF = 0, dacă nu este egalitate (d) <> (s)  
**JL / JNGE** SF <> OF, dacă (d) < (s), pentru numere cu semn  
**JLE / JNG** SF <> OF sau ZF = 1, (d)<=(s), numere cu semn  
**JNL / JGE** SF = OF, dacă (d) >= (s), pentru numere cu semn  
**JNLE / JG** SF = OF și ZF = 0, (d) > (s), pentru numere cu semn  
**JB / JNAE / JC** CF = 1, dacă (d) < (s), numere fără semn  
**JBE / JNA** CF = 1 sau ZF = 1, (d)<=(s), numere fără semn  
**JNB / JAE / JNC** CF = 0, dacă (d) >= (s), numere fără semn  
**JNBE / JA** CF = 0 și ZF = 0, (d) > (s), pentru numere fără semn  
**JP / JPE** PF = 1, dacă numărul are paritate pară  
**JNP / JPO** PF = 0, dacă numărul are paritate impară  
**JO/JNO** OF = 1/0, dacă este/ nu este depășire de reprezentare  
**JS /JNS** SF = 1/0, dacă numărul este negativ/ pozitiv  
**J(E)CXZ** (E)CX = 0, test asupra registrului ECX sau CX

condiție	ZF	CF	condiție OF	SF	ZF	
d > s	0	0	d > s	0/1	0/1	0
d = s	1	0	d = s	0	0	1
d < s	0	1	d < s	0/1	1/0	0

*pentru numere fără semn*                      *pentru numere cu semn*

<i>salt pentru</i>	<i>numere fără semn</i>	<i>numere cu semn</i>
d = s	<b>JE / JZ</b>	<b>JE / JZ</b>
d <> s	<b>JNE / JNZ</b>	<b>JNE / JNZ</b>
d < s	<b>JB / JNAE / JC</b>	<b>JL / JNGE</b>
d > s	<b>JA / JNBE</b>	<b>JG / JNLE</b>
d <= s	<b>JBE / JNA</b>	<b>JLE / JNG</b>
d >= s	<b>JAE / JNB / JNC</b>	<b>JGE / JNL</b>

Determinarea valorii maxime dintr-un șir de valori (cu semn).

**.model small**

**.stack 10h**

**.data**

```

sir    db    10,20,-30,100,-100,200
lung  dw    $-sir
maximdb  ?
mes_sir_vid db    'sir vid de valori$'
```

```

.code
mov ax, @data ; inițializare registru segment
mov ds, ax
mov cx, lung ; contor număr de valori din șir
jcxz sir_vid ; dacă șirul este vid se tipărește mesaj
lea si, sir ; index elemente din șir
cld ; direcția de parcurgere
mov bl, [si] ; se inițializează valoarea maximă din șir
inc si ; actualizare index elemente
dec cx ; actualizare contor
jcxz gata ; dacă a fost un singur element s-a terminat
iar: lodsb ; citește element curent din șir
cmp bl, al ; se compară cu maximul curent
jge urm ; dacă max > val. curentă trece la elem. următor
; pentru numere fără semn se înlocuiește jge cu jnc
mov bl, al ; altfel se actualizează valoarea maximă
urm: loop iar ; se reia ciclul dacă mai sunt elem. în șir
gata: mov maxim, bl ; se depune valoarea maximă
iesire: mov ax, 4c00h ; revenire DOS
int 21h
sir_vid:
lea dx, mes_sir_vid ; se tipărește mesajul
mov ah, 9 ; 'sir vid de valori'
int 21h
jmp iesire ; revenire DOS
end

```

### Instrucțiuni de ciclare

Aceste instrucțiuni pot fi folosite pentru a controla repetiția unor cicluri soft. Ele permit o programare ușoară a structurilor de control de tip ciclu cu test la sfârșit. Ele utilizează drept contor, al ciclului, registrul CX. Instrucțiunile de ciclare sunt auto-relative, și pot transfera controlul la o locație (țintă) aflată în intervalul [-128,+127] de octeți față de ele, deci realizează numai transferuri de tip SHORT (atât pentru 286 cât și pentru 386/486). Aceste instrucțiuni nu modifică nici un indicator.

**LOOP <short\_etich>** (LOOP control with CX counter)  
 $(CX) \leftarrow (CX) - 1$   
if  $(CX) < 0$  then  
 $(IP) \leftarrow (IP) + \text{deplasament}(\text{short\_etich});$

1) Calculul sumei elementelor unui șir de tip cuvânt.

```

mov cx, lung_sir ; contor număr de elemente
mov ax, 0 ; suma va fi pe două cuvinte
mov bx, ax ; și se inițializează cu 0
mov si, ax ; index pentru adresarea elementelor
aduna:

```

```

add ax, sir[si]
adc bx, 0 ; transporturile se acumulează în BX
add si, type sir ; actualizare index
loop aduna ; dacă nu e gata se reia adunarea
mov suma, ax ; se depune rezultatul, începând cu
mov suma[2], bx ; cu cuvântul mai puțin semnificativ

```

2) Determinarea primelor  $N$  numere din șirul lui Fibonacci.

```

mov ax, 1 ; se inițializează primele două valori:0,1
mov bx, 0 ; formula este: fib(n)=fib(n-1)+fib(n-2)
mov cx, N ; numărul de valori calculate
mov di, bx ; indexul adresei unde se depun numerele
urm: mov si, ax ; se salvează fib(n-1), care va fi fib(n-2)
add ax, bx ; se calculează fib(n), care va fi fib(n-1)
mov bx, si ; se actualizează fib(n-2)
mov fibonacci[di], ax ; se depune fib(n)
add di, type fibonacci ; actualizare index
loop urm ; se reia dacă nu s-au calculat N numere
<short_etich>

```

(LOOP while Equal/Zero, with CX counter)

```

(CX) ← (CX) - 1
if ((CX) <> 0) and (ZF=1) then
    (IP) ← (IP) + deplasament (short_etich);

```

Determinarea primului element diferit de 0, dintr-un șir.

```

mov cx, lung_sir ; contorul șirului
mov si, -1 ; inițializare SI
reia: inc si
cmp sir[si], 0 ; se compară cu 0
loope reia ; dacă este 0 se reia comparația
jz sir_zero ; dacă ZF=1, tot șirul conține numai 0
aici: ; s-a determinat primul element <> 0, la adresa SI

```

LOOPNE / LOOPNZ <short\_etich>

(LOOP while Not Equal/Zero, with CX counter)

```

(CX) ← (CX) - 1
if ((CX) <> 0) and (ZF=0) then
    (IP) ← (IP) + deplasament (short_etich);

```

1) Determinarea ultimului element dintr-o listă înlănțuită. Ultimul element dintr-o listă va conține valoarea zero în câmpul de adresă.

```

lea bx, offset cap_lista ; adresa de început a listei
mov cx, N ; dimensiunea maximă a listei
urm: mov bx, [bx + lung_info] ; adresa elementului următor
cmp bx, 0 ; se compară cu 0
loopne urm ; se reia dacă nu s-a găsit
je gasit ; s-a găsit ultimul element din listă, adr. în BX

```

**; dacă nu se execută saltul → nu s-a găsit ultimul element printre cele N elemente**

2) Determinarea primei valori egală cu zero, dintr-un șir de valori.

```
mov  cx, lung_sir  ; contorul șirului
mov  si, -1        ; inițializare SI, pentru a nu afecta
reia: inc  si      ; indicatorii, înaintea instrucțiunii LOOP
      cmp  sir[si], 0 ; se compară cu 0
      loopne reia   ; dacă nu este 0 se reia comparația
      jnz  sir_zero ; dacă ZF=0, tot șirul conține numai valori <> 0
aici:                ; s-a determinat primul element = 0, la adresa SI
```

### Instrucțiuni de întrerupere

Aceste instrucțiuni permit rutinelor de servire a întreruperilor să fie activate prin program la fel ca la apariția unei întreruperi hardware externe. Singura instrucțiune ce alterează indicatorii este IRET.

**INT** <tip\_întrerupere> (call to INTerrupt procedure)

```
(SP) ← (SP) - 2
((SP)+1:(SP)) ← indicatori
(tf), (if) ← 0
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (CS)
(CS) ← (n * 4 + 2)
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (IP)
(IP) ← (n * 4)
```

**INTO** (INTerrupt on Overflow)

Această instrucțiune generează întrerupere dacă OF=1.

**BOUND** <dest\_reg>, <sursa\_mem>

(check array index against BOUNDS)

Determină dacă valoarea, cu semn, conținută în registrul destinație (16/32 biți), specificat de instrucțiune, se află între limitele specificate de operandul sursă, care trebuie să fie operand din memorie (16/32 biți). Se compară operandul destinație cu două cuvinte (sau cuvinte duble), aflate în memorie de la adresa specificată de cel de-al doilea operand. Dacă valoarea destinației nu este între cele două limite (la locații succesive), specificate de sursă, din memorie, se generează întrerupere pe nivelul 5 (INT 5).

**IRET / IRETD** (Interrupt RETurn)

Transferă controlul înapoi la punctul de întrerupere, prin refacerea IP, CS și a indicatorilor, din stivă. Astfel instrucțiunea IRET afectează toți indicatorii, prin restaurarea lor, la valorile anterior salvate.

Împărțirea unui număr de 32 biți ( $A_1, A_0$ ) la un număr de 16 biți (B). Rezultatul, câtul, poate avea, cel mult, 32 de biți ( $Q_1, Q_0$ ), iar restul 16 biți ( $R_0$ ).

Se va încerca mai întâi o împărțire obișnuită, și dacă câtul este de 16 biți, operația este terminată. Dacă nu, deci dacă câtul este de 32 de biți atunci se va face împărțirea în modul următor:

$$A_1 * 2^{16} + A_0 = (Q_1 * 2^{16} + Q_0) * B + R_0$$

unde  $Q_1 = [ A_1 / B ]$

și deci  $A_1 = Q_1 * B + R_1$

înlocuind  $A_1$  în prima relație rezultă:

$$R_1 * 2^{16} + A_0 = Q_0 * B + R_0$$

Deci ordinea operațiilor va fi:

- se împarte  $A_1$  la  $B$  și rezultă:  $Q_1$  și  $R_1$ ;

- se împarte  $(R_1 * 2^{16} + A_0)$  la  $B$  și rezultă:  $Q_0$  și  $R_0$ .

; **procedura de împărțire, primește:**

; **(DX, AX) - deîmpărțit (32 biți)**

; **(BX) - împărțitor (16 biți)**

; **și va returna:**

; **(BX, AX) - câtul**

; **(DX) - restul**

**pdiv**

**proc**

**cmp bx, 0 ; împărțitor = 0 ?**

**jnz cdiv ; dacă nu se continuă împărțirea, altfel**

**int 0 ; se apelează procedura de eroare la împărțire**

**jmp eroare**

**cdiv: push es ; se salvează registrele de lucru**

**push di**

**push cx**

**mov di, 0**

**mov es, di ; ES:DI=0:0, adresa întrerupere nivel 0**

**push es:[di] ; se salvează adresa de tratare întrerupere**

**push es:[di+2] ; nivel 0 (eroare la împărțire) în stivă**

**; se înlocuiește adresa salvată (proc. de eroare la împărțire) cu**

**; adresa procedurii prezentate anterior**

**lea cx, int\_0 ; offset procedura de împărțire**

**mov es:[di], cx ; se depune în locul proc. de eroare**

**mov cx, cs ; adresa de segment a noii proceduri**

**mov es:[di+2], cx ; se pune în locul celei de eroare**

**; se încearcă împărțirea normală**

**div bx ; dacă câtul are 16 biți s-a terminat și BX = 0**

**; dacă nu, deci dacă câtul are 32 biți, se va genera întrerupere**

**; pe nivelul 0, care a fost înlocuită cu procedura de împărțire**

**xor bx, bx ; BX = 0, câtul este doar în AX**

**revenire: pop es:[di+2] ; se reface adresa procedurii**

**pop es:[di] ; de întrerupere, nivel 0, eroare împărțire**

**pop cx ; se refac registrele salvate în stivă**

**pop di**

**pop es**

**ret**

**; procedura propriu-zisă de împărțire, care a fost pusă în locul**

```

; procedurii de tratare eroare la împărțire (nivel 0)
int_0: push bp      ; salvare BP, pentru a accesa stiva
      mov bp, sp    ; se va înlocui adresa salvată, a instr.
      ; următoare împărțirii cu cea a instrucțiunii de 'revenire'
      mov word ptr [bp+2], offset revenire
      ; sau, întrucât instr. 'xor bx, bx' are doi octeți.
      ; se poate utiliza și: add word ptr [bp+2], 2
      push ax       ; se salvează A0
      mov ax, dx    ; (AX) ← A1
      sub dx, dx    ; pregătire de împărțit (DX=0)
;      se realizează împărțirea A1/B -> (R1,Q1) = (DX,AX)
      div bx
      pop cx        ; CX = A0
      push ax       ; se salvează Q1
      mov ax, cx    ; (DX,AX) = (R1,A0)
;      se realizează împărțirea (R1,A0)/B -> (R0,Q0) = (DX,AX)
      div bx       ; (DX,AX) = (R0,Q0)
      pop bx       ; se reface câțul: (BX) = Q1
      pop bp
      iret
eroare: ..... ret
pdiv  endp

```

### Instrucțiuni de control procesor

Aceste instrucțiuni permit programelor să controleze diferite funcțiuni ale procesorului. Un grup de instrucțiuni actualizează indicatorii, altul este utilizat pentru sincronizarea microprocesorului cu evenimente externe, și mai există o instrucțiune care nu realizează nimic (NOP).

#### Instrucțiuni pentru poziționarea indicatorilor

**CLC** (CLear Carry flag)

CF ← 0

**STC** (SeT Carry flag)

CF ← 1

**CMC** (CoMplement Carry flag)

CF ← not (CF)

Aceste instrucțiuni de poziționare a lui CF se utilizează la operațiile de rotație, deplasare sau la adunări/scăderi de numere reprezentate pe mai mulți octeți/cuvinte etc.

**CLD** (CLear Direction flag)

DF ← 0

**STD** (SeT Direction flag)

DF ← 1

Acest indicator va determina direcția de parcurgere a șirurilor.

**CLI** (CLear Interrupt enable flag)

IF ← 0

**STI** (SeT Interrupt enable flag)



IF ← 1

Indicatorul IF dezactivează sau activează, prin poziționare pe 0, respectiv 1, întreruperile externe, ce apar pe linia INTR. Sunt dezactivate numai întreruperile mascabile (INTR), nu și întreruperea nemascabilă NMI, care este servită, indiferent de poziția lui IF. La activarea sistemului de întreruperi (STI), întreruperea, pe linia INTR, este recunoscută după execuția instrucțiunii următoare lui STI.

### Instrucțiuni de sincronizare cu evenimente externe

#### HLT (HaLT)

Determină intrarea proc. în starea 'halt', din care iese doar dacă:

- se activează linia RESET;
- se transmite semnal de întrerupere pe linia NMI;
- se transmite semnal de întrerupere pe linia INTR, dacă întreruperile nu au fost dezactivate.

dezactivate.

Această instrucțiune se poate folosi la sfârșitul unei bucle soft, în situația în care programul trebuie să aștepte un eveniment extern (întrerupere de la un dispozitiv extern). În acest caz se va salva în stivă adresa instrucțiunii următoare lui HLT.

#### WAIT (WaIT)

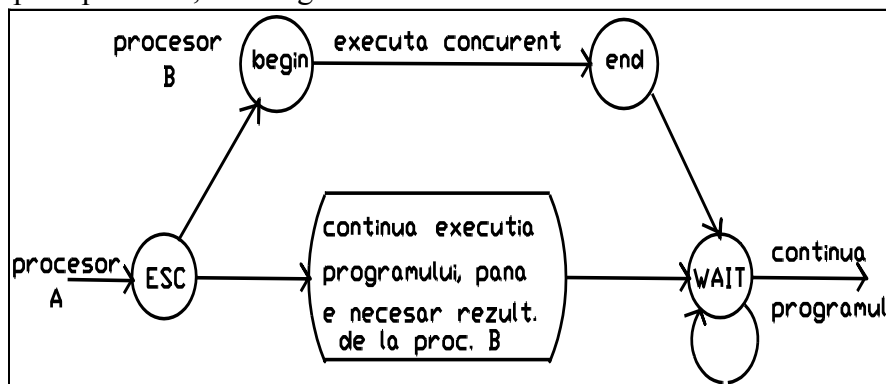
Determină intrarea proc. în starea 'wait', cât timp linia BUSY\ este activă. Procesorul intră într-o stare 'idle', și repetă testarea liniei BUSY\, ciclic la intervale de șase perioade de ceas. Când linia BUSY\ devine inactivă, se continuă execuția programului cu instrucțiunea următoare lui WAIT.

Această stare poate fi întreruptă, printr-un semnal de întrerupere, dar după tratarea întreruperii se reîntră în această stare, întrucât se salvează în stivă adresa acestei instrucțiuni.

#### ESC <Cod\_Op>, <sursa> (ESCaPe)

Această instrucțiune este specifică coprocesorului matematic, și ea este identificată prin primii cinci biți ai codului de operare care sunt 11011; ceilalți biți din câmpul codului operației identifică tipul operației coprocesorului. De fapt coprocesorul urmărește magistrala sistemului, și când identifică execuția acestei instrucțiuni, el o capturează; dacă operandul sursă este un registru procesorul nu face nimic cu el, iar dacă este un operand din memorie, îl citește dar îl ignoră. În schimb coprocesorul poate captura acest operand când este citit din memorie.

Când ESC se utilizează în conjuncție cu WAIT, se poate iniția o operație care se execută concurrent pe coprocesor, ca în figura următoare.



**LOCK**

(assert LOCK# signal prefix)

Acesta este un prefix, de un octet, care determină activarea semnalului de magistrală LOCK, cât timp se execută instrucțiunea precedată de acest prefix. Într-un sistem multiprocesor, acest semnal poate fi utilizat pentru a asigura procesorului acces exclusiv la utilizarea magistralei, și deci a unei resurse (memorii) partajate.

**mov al, 1****astept:**

```
lock  xchg  al, semafor    ; citire și setare semafor
test  al, al                ; testare semafor = 0 ?,
jnz   astept                ; dacă nu, resursa este ocupată
                                ; și se așteaptă eliberarea ei
....  .                    ; utilizare exclusivă a resursei
mov   semafor, 0            ; eliberare resursă
```

**NOP**

(No OPeration)

După cum spune și numele acestei instrucțiuni, ea nu realizează nici o prelucrare; nu afectează nici un indicator. Ea durează trei perioade de ceas, și are codul mașină al instrucțiunii XCHG AX,AX, sau XCHG EAX,EAX.