Analiza instrucţiunii JMP. Salturi NEAR şi salturi FAR.

Exemplul 4.3.1.2. Prezentăm în continuare un exemplu edificator pentru modul de transfer al controlului la o etichetă, punând în evidenţă deosebirile dintre un transfer *direct* şi unul *indirect*.

```
segment data
        aici   DD  here          ;echivalent cu aici := offsetul etichetei here din segmentul de cod

segment code
        .       .       .
        mov   eax, [aici]       ;se încarcă în EAX conţinutul variabilei aici (adică deplasamentul lui here în
                                  cadrul segmentului code – echivalent deci ca efect cu:   mov eax, here)

        mov   ebx, aici         ;se încarcă în EBX deplasamentul lui aici în cadrul segmentului data
                                ; (probabil 00401000h)
```

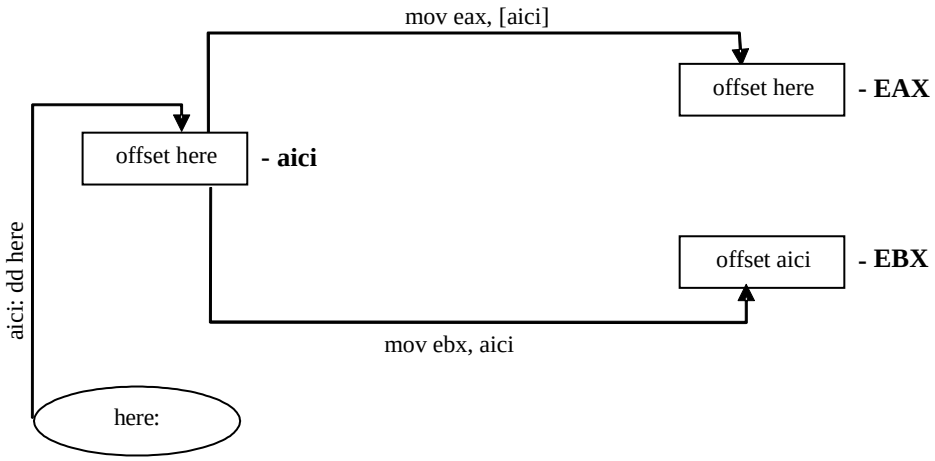

Fig. 4.4. Iniţializarea variabilei aici şi a regiştrilor EAX şi EBX.

```
jmp   [aici]    ;salt la adresa desemnată de valoarea variabilei aici (care este adresa lui here), deci
                  instrucţiune echivalentă cu jmp here  ; ce face jmp aici ??? - acelasi lucru ca si
                  jmp ebx ! salt la CS:EIP cu EIP=offset (aici) din SEGMENT DATA (00401000h) ; salt la niste instr. care merg pana la primul access violation

jmp   here      ;salt la adresa lui here (sau, echivalent, salt la eticheta here); jmp [here] ?? – JMP
DWORD PTR DS:[00402014] – cel mai probabil Access violation....


jmp eax         ;salt la adresa conţinută în EAX (adresare registru în mod direct), adică la here
                ; ce face prin comparatie jmp [eax] ???  JMP DWORD PTR DS:[EAX] – cel mai
                ; probabil Access violation....

jmp   [ebx]     ;salt la adresa conţinută în locaţia de memorie a cărei adresă este conţinută în ;EBX
                  (adresare registru în mod indirect - singura situaţie de apel indirect din ;acest
                  exemplu) – ce face prin comparatie jmp ebx ???  -  salt la CS:EIP cu
                  EIP=offset (aici) din SEGMENT DATA (00401000h) ; salt la niste instr. care
                  merg pana la primul access violation
```

;în EBX se află adresa variabilei aici, deci se accesează conţinutul acestei variabile. În această locaţie de memorie se găseşte offset-ul etichetei here, deci se va efectua saltul la adresa here - **ca urmare, ultimele 4 instrucţiuni de mai sus sunt toate echivalente cu jmp here**

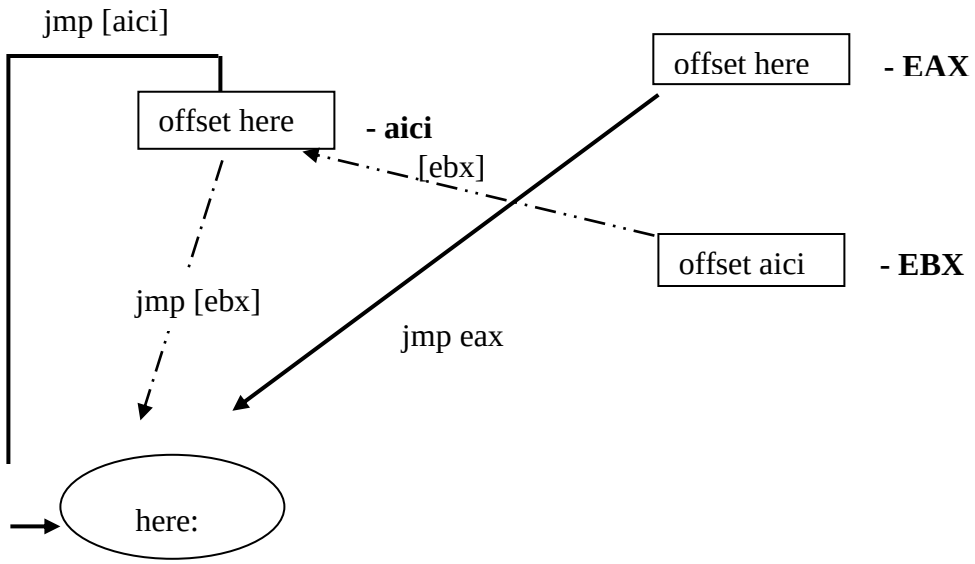

Fig. 4.5. Modalităţi alternative de efectuare a salturilor la eticheta *here*.

```
        jmp   [ebp]   ;  JMP DWORD PTR SS:[EBP]
.....................................................................
        here:
              mov ecx, 0ffh
```

Explicaţii asupra interactiunii dintre regulile de asociere implicita a unui offset cu registrul segment corespunzător şi efectuarea corespunzătoare a saltului la offset-ul precizat

JMP [var_mem] ; JMP DWORD PTR DS:[00401???]

JMP [EBX] ; JMP DWORD PTR DS:[EBX]
JMP [EBP] ; JMP DWORD PTR SS:[EBP]
JMP here ; JMP DWORD PTR DS:[00401014] - va face saltul DIRECT la offsetul respectiv in code segment

conform regulilor de asociere implicită a unui offset cu registrul segment corespunzător.

In acest caz să ne aşteptăm ca saltul să se efectueze în data segment la offset-ul respectiv si sa interpreteze ceea ce gaseste acolo drept coduri de instructiuni (avand in vedere raportarea offset-ului la DS) ??. Adica adresa FAR de salt sa fie DS:[00401000] si respectiv DS:[reg] ?

NU ! Nu se întâmplă aşa ceva. In primul rând tipul de salt implicit este NEAR (a se vedea în acest sens cum genereaza asamblorul echivalentul obiect al instructiunilor prin DWORD PTR, deci ia în considerare DOAR OFFSET-ul respectiv - raportat la DS !). Fiind vorba depre instructiuni de salt NEAR acestea se efectuează IN ACELASI SEGMENT DE COD în care apar cele 2 instructiuni, ca urmare salturile se efectuează de fapt la CS:[var_mem] si respectiv CS:[reg].

Deci, un salt NEAR chiar daca este prefixat implicit cu DS nu face salt in data segment, din data segment doar incarca offsetul inspre care sa faca salt in segmentul de cod. **Prefixarea este utila aici doar pentru a permite incarcarea valorii OFFSET-ului pointer-ului,** care ar putea ipotetic stocat in orice segment.

Ca urmare, chiar daca folosim prefixarea explicită, de exemplu

jmp [ss: ebx + 12]
jmp [ss:ebp +12] equiv with jmp [ebp +12]

saltul fiind în continuare NEAR se va efectua în cadrul aceluiaşi segment de cod, adică la CS : EIP, adică la CS : [valoare offset luata din stiva]

Deci prefixarea cu un registru segment conform regulilor implicite are doar rolul de a indica segmentul corect din care sa se incarce offset-ul catre care se va efectua saltul NEAR in cadrul segmentului de COD curent !

Dacă dorim însă efectuarea saltului într-un segment diferit (salt FAR) trebuie să specificăm explicit acest lucru prin intermediul operatorului de tip FAR:

jmp far [ss: ebx + 12] => CS : EIP <- adresa far (48 biti) luati din stack segm.

(9b 7a 52 61 c2 65) → EIP = 61 52 7a 9b ; CS= 65 c2
„Mov CS:EIP, [adr]" ;

Operatorul FAR precizeaza aici faptul ca nu doar EIP trebuie populat cu ce se afla la adresa de memorie indicata ci si CS-ul trebuie încărcat cu o nouă valoare.

Pe scurt avem:
- valoarea pointer-ului fi stocata oriunde in memorie, rezultand ca orice specificare de adresa valida la un mov poate aparea la fel de bine si la jmp (de exemplu **jmp [gs:ebx + esi*8 - 1023]**)
- pointer-ul in sine (deci octetii luati de la respectiva adresa de memorie) poate fi far sau near, fiind, dupa caz, aplicat fie lui doar lui EIP (dacă e NEAR), fie perechii CS:EIP daca saltul e FAR.

Saltul poate fi imaginat ca fiind echivalent, ipotetic, cu instructiuni MOV de forma:
- jmp [gs:ebx + esi * 8 - 1023] <=> mov EIP, [gs:ebx + esi * 8 - 1023]
- jmp FAR [gs:ebx + esi * 8 - 1023] <=> mov EIP, DWORD [gs:ebx + esi * 8 - 1023] +
mov CS, WORD [gs:ebx + esi * 8 - 1023 + 4]

La adresa [gs:ebx + esi * 8 - 1023] am gasit in exemplul luat de mine configuratia de memorie:
7B 8C A4 56 D4 47 98 B7.....

Mov CS:EIP, [memory] → EIP = 56 A4 8C 7B
CS = 47 D4

JMP prin etichete – always NEAR !

```
segment data use32 class=DATA

   a db 1,2,3,4
    start3:
     mov edx, eax
      ........

segment code use32 class=code
   start:
         mov edx, eax
         jmp start2  - ok – salt NEAR  - JMP  00403000 (offset code segment = 00402000)
         jmp start3 – ok – salt NEAR –   JMP  00401004 (offset data segment = 00401000)

         jmp far start2  - Segment selector relocations are not supported in PE file – syntax error !
         jmp far start3  - Segment selector relocations are not supported in PE file– syntax error !

         ;Cele doua salturi de mai sus jmp start2 si respectiv jmp start3 se vor efectua la etichetele
         ;mentionate, start2 si start3 insa ele nu vor fi considerate salturi FAR (dovada este ca
         precizarea ;acestui atribut mai jos in celelalte doua variante ale instructiunilor va furniza
         syntax error !)
         ;Ele vor fi considerate salturi NEAR datorita modelului de memorie FLAT utilizat de catre SO

         add eax,1
   final:
     push    dword 0
     call    [exit]


 segment code1 use32 class=code

    start2:
       mov eax, ebx

    push dword 0
    call [exit]
```

De ce ?... Din cauza **"Flat memory model"**

Concluzii finale.

- **Salturi NEAR – se pot realiza prin oricare dintre cele 3 tipuri de operanzi (eticheta, registru, operand cu adresare la memorie)**

- **Salturi FAR (asta insemnand modificarea si a valorii din CS, nu numai a valorii din EIP) – se pot realiza DOAR prin intermediul unui operand adresare la memorie pe 48 de biti (pointer FAR). De ce doar asa si prin etichete sau registri nu ?**

- **Prin etichete, chiar daca se sare intr-un alt segment nu se considera ca este salt FAR deoarece nu se modifica CS-ul (din cauza modelului de memorie implementat – Flat Memory Model). Se va modifica doar EIP-ul si saltul se considera dpdv tehnic ca fiind un salt NEAR.**

- **Prin registri nu este posibil deoarece registri sunt pe 32 de biti si se poate astfel specifica drept operand al unui JMP doar un offset (salt NEAR), deci practic suntem in imposibilitatea de a preciza un salt FAR cu un operand limitat la 32 de biti.**