JMP instruction analysis. NEAR and FAR jumps.

We present below a very relevant example for understanding the control transfer to a label, highlighting the differences between a direct transfer vs. an indirect one.

```
segment data
        aici   DD  here        ;equiv. with  aici := offsetul of label here from code segment

segment code
        .       .       .
        mov   eax, [aici]       ;we load EAX with the contents of variable aici (that is the offset of here
                                  inside the code segment – same effect as  mov eax, here)

        mov  ebx, aici          ;we load EBX with the offset of aici inside the data segment
                                ; (probable 00401000h)
```

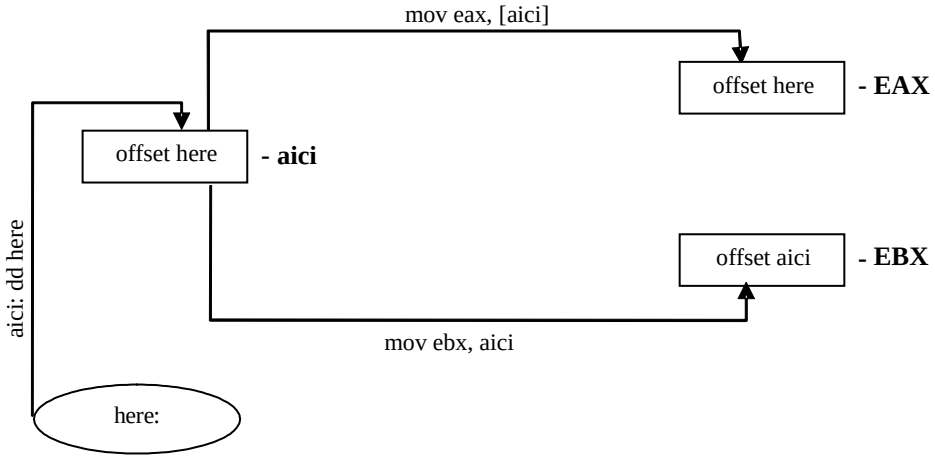

Fig. 4.4. Initializing variable aici and registers EAX and EBX.

jmp [aici] ;jump to the address indicated by the value of variable aici (which is the address of here), so this is an instruction equiv with jmp here ; what does in contrast **jmp aici** ??? – the same thing as jmp ebx ! **jump to CS:EIP with EIP=offset (aici) from SEGMENT DATA (00401000h)** ; jump to some instructions going until the first „access violation"

jmp here ;jump to the address of here (or, equiv, jump to label here); **jmp [here]** ?? – JMP DWORD PTR DS:[00402014] – most probably „Access violation"....

jmp eax ;jump to the address indicated by the contents of EAX (accessing register in direct mode), that is to label here ; in contrast, what does **jmp [eax]** ??? JMP DWORD PTR DS:[EAX] – most probably „Access violation"....

jmp [ebx] ;jump to the address stored in the memory location having the address the contents of EBX (indirect register access – the only indirect access from this example) – what does in contrast **jmp ebx** ??? - jump to **CS:EIP with EIP=offset (aici) from the SEGMENT DATA (00401000h)** ; jump to some instructions going until the first „access violation"

;in EBX we have the address of variable aici, so the contents of this variable will be accessed. In this memory location we find the offset of label here, so a jump to address here will be performed - **consequently, the last 4 instructions from above are all equivalent to jmp here**

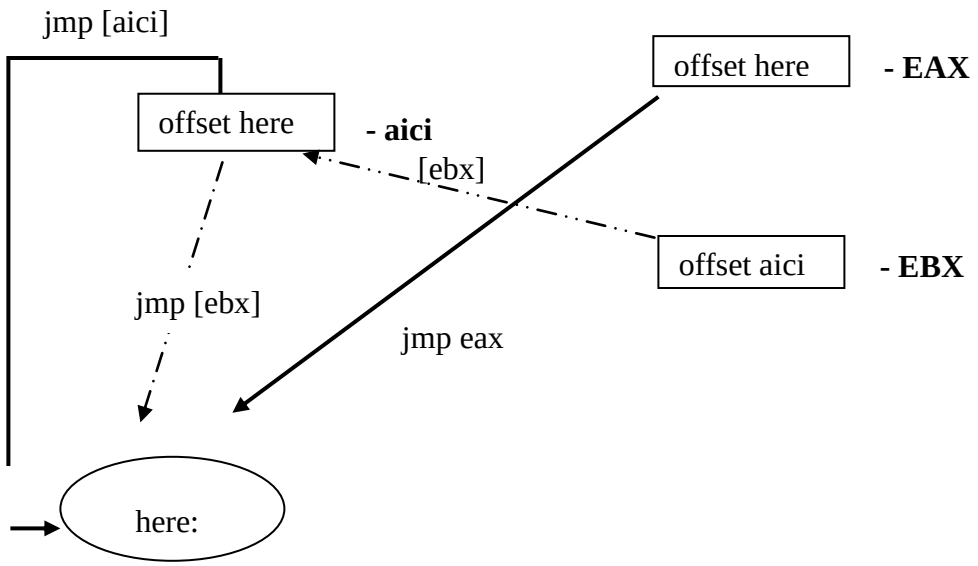

Fig. 4.5. Alternative ways for performing jumps to the label *here.*

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

Explanations on the interaction between the implicit association rules of an offset with the corresponding segment register and performing the corresponding jump to the specified offset

JMP [var_mem] ; JMP DWORD PTR DS:[00401704] - NEAR jump to offset from DS:[00401704]
JMP [EBX] ; JMP DWORD PTR DS:[EBX] - NEAR jump to offset from DS:[EBX]
JMP [EBP] ; JMP DWORD PTR SS:[EBP] - NEAR jump to offset from SS:[EBP]

JMP here ; JMP [SHORT] 00402024 - va face saltul DIRECT la offsetul respectiv in code segment

Accordingly to the implicit association rules of an offset with the corresponding segment register

The INDIRECT addressing operand after JMP tells us FROM WHERE to take the OFFSET to perform the NEAR jump to (jump within the current code segment). The last instruction expresses a DIRECT jump to the offset computed as the value associated with the here label (jump to CS:here).

Even if we use explicit prefixing with a segment register of the destination operand, the jump will NOT be a FAR one. **FAR will only be the ADDRESS from which the OFFSET will be taken where the NEAR jump will be made.**

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

The jump will still remain a NEAR one and will be made in the same code segment, namely to
CS : offset value taken from from SS:[ebp+12]

If we want to perform the jump in a different segment (FAR jump) we must explicitly specify this by means of the FAR type operator, which will impose the destination operand of the JMP instruction to be treated as a FAR ADDRESS:

jmp far [ebx + 12] => CS : EIP <- far address (48 bits = 6 bytes) equiv. to
jmp far [DS: ebx + 12] => CS : EIP <- far address (48 bits = 6 bytes)

(9b 7a 52 61 c2 65) → EIP = 61 52 7a 9b ; CS= 65 c2

„Mov CS:EIP, [adr_far]" ;

Or by explicit prefixing: **jmp far [ss: ebx + 12] => CS : EIP <- adresa far (48 biţi)**

The FAR operator specifies here that not only EIP must be populated with what is at the memory address indicated by the destination operand (NEAR address = offset), but also the CS must be loaded with a new value (CS:EIP = FAR address).

In short and as a conclusion we have :
- the value of the pointer representing the address where the jump has to be made can be stored anywhere in memory, this meaning that any offset specification that is valid for a MOV instruction can be also present as an operator for a JMP instruction (for example **jmp [gs:ebx + esi*8 - 1023]**)
- the contents of the pointer (the bytes taken from that specific memory address) may be near or far, depending on how the programmer specifies or not the FAR operator, being thus applied either only to EIP (if the jump is NEAR), either to the pair CS:EIP if the jump is FAR.

Any jump can be considered hypothetically equivalent to MOV instructions such as:
- jmp [gs:ebx + esi * 8 - 1023] <=> "mov EIP, DWORD [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]"

At the address [gs:ebx + esi * 8 - 1023] we found for the concrete example the following values:

7B 8C A4 56 D4 47 98 B7.....

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

JMP through labels – always NEAR !

```
segment data use32 class=DATA

    a db 1,2,3,4
     start3:
      mov edx, eax    ok ! – the control is transferred at start3 and this instruction is executată !


segment code use32 class=code        offset code segment = 00402000
    start:
          mov edx, eax
          jmp start2  - ok – NEAR jmp -   JMP  00403000 (offset code1 segment = 00403000)
          jmp start3 – ok – NEAR jmp –   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 !

          ;The two jumps above jmp start2 and jmp start3 respectively will be done to the specified
          ;labels, start2 and start3 but they will not be considered FAR jumps (the proof is that the
          ;specification of this attribute in the other two variants of the instructions from above will
          ;lead to a syntax error!)

          ;They will be considered NEAR jumps due to the FLAT memory model used by the OS

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


 segment code1 use32 class=code

    start2:
        mov eax, ebx

    push dword 0
    call [exit]
```

Why ?... Because of the **"Flat memory model"**

Final conclusions.

- **NEAR jumps – can be accomplished through any of the three operand types (label, register, memory addressing operand)**

- **FAR jumps (this meaning modifying also the CS value, not only that from EIP) – can be performed ONLY by a memory addressing operand on 48 bits (pointer FAR). Why only so and not also by labels or registers ?**

- **- if we would have used labels, even if we jump into another segment (an action possible as you can see above) this is not considered a FAR jump because CS is not modified (due to the implemented memory model – Flat Memory Model). Only EIP will be changed and the jump is technically considered to be a NEAR one.**

- **- if we would have used registers as operands we may not perform a far jump, because registers are on 32 bits and we may so specify maximum an offset (NEAR jump), so we are practically in the case when it is impossible to specify a FAR jump using only a 32 bits operand.**