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)
```

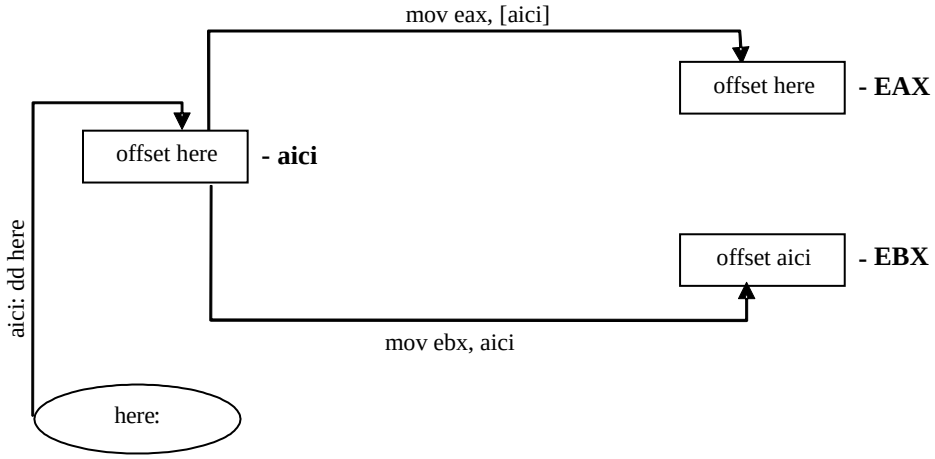

Fig. 4.4. Initializing variable aici and registers EAX and EBX.

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**

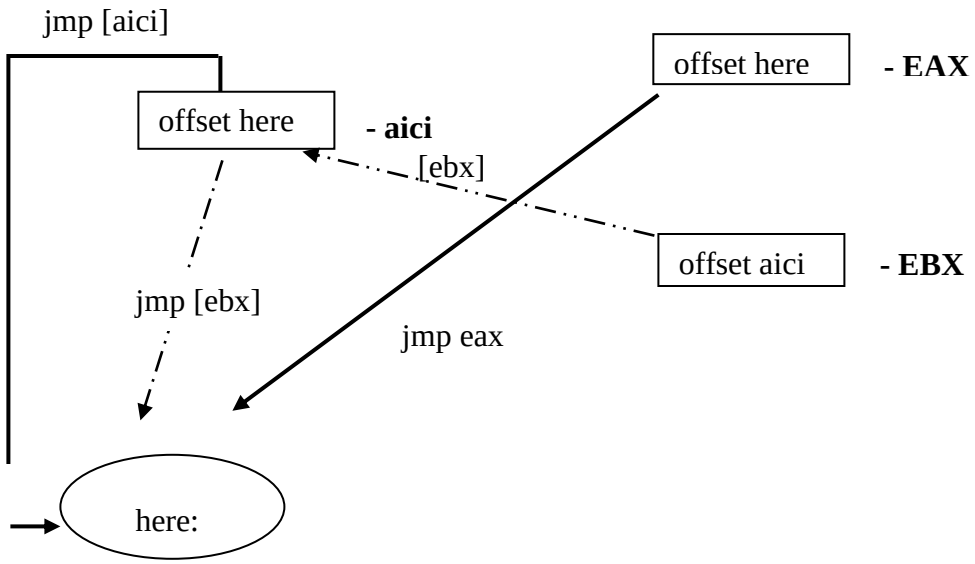

Fig. 4.5. Alternative ways for performing jumps to the label *here.*

```
        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:[004010??]**

JMP [EBX] **; JMP DWORD PTR DS:[EBX]**
JMP [EBP] **; JMP DWORD PTR SS:[EBP]**

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

In this case do we have to expect that the jump to be made in the data segment at the specified offset and the processor to interpret that data as instructions code ? (taking into consideration the implicit offset positioning relative to DS) ??. That is, the FAR jumping address to be DS:[00401000] and DS:[reg] respectively ?

NO ! NOT AT ALL ! Something like that doesn't happen. First of all, the IMPLICIT jump type is NEAR (check in OllyDbg how NASM is generating the corresponding object code by DWORD PTR, so it takes into consideration JUST the OFFSET - relative to DS or SS !). Being NEAR jump instructions, these will be made IN THE SAME MEMORY (CODE) SEGMENT in which those three instructions appear ,so the jumps will be made in fact to CS:[var_mem] and CS:[reg] respectively.

So... a NEAR jump, even if it is implicitly prefixed with DS or SS doesn't make a jump into the data segment or in the stack segment! From that segment it will only **load the corresponding offset** to which to perform the jump inside the current CODE SEGMENT !!! That is why we call it in fact a NEAR JUMP !!!... isn't it ? **Prefixing is useful here only for allowing to load the corresponding OFFSET value of the pointer,** offset which can be placed/present/specified/stored in any (other) segment.

It follows that even if we use explicit prefixing, like for example

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

because the jump is still a NEAR one, it will be made inside the same (current) code segment, that is to CS : EIP, namely to CS : offset value taken from the stack

So, prefixing a target of a jump with a segment register has only the task of correctly indicate the actual segment from which the required offset will be loaded, offset to which the NEAR jump will be performed inside the current code segment !

If we wish to make a jump to a different segment (namely to perform a FAR jump) we must EXPLICITLY specify that by using the FAR type operator:

jmp far [ss: ebx + 12] => CS : EIP <- the far address (48 bits) taken from the stack segm.

The FAR operator specifies here that not only EIP must be loaded with what we have at the specified address, but also CS must be loaded with a new value (this makes in fact the jump to be a FAR one).

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]"

[gs:ebx + esi * 8 - 1023]

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   (to be verified !!!)  - !!!!

segment code use32 class=code
   start:
         mov edx, eax
         jmp start2  - ok – NEAR jmp -   JMP  00403000 (offset code segment = 00402000)
         jmp start3 – ok – NEAR jmp –   JMP  00401004 (offset data segment = 00401000)
         jmp far start2  - Segment selector relocations are not supported in PE file
         jmp far start3  - Segment selector relocations are not supported in PE file

         ;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 below in the other two variants of the instructions 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 all 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 ?**
- **- by 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.**
- **Using 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.**