Location counter and pointer arithmetic

Location counter and pointer arithmetic

SEGMENT data

a db 1,2,3,4 ; 01 02 03 04

lg db $-a ; 04
lg0 db $-data ; syntax error – expression is not simple or relocatable
in TASM will be the same effect as above because in TASM, MASM
offset(segment_name)=0 !!!; in NASM NO WAY, offset(data) = 00402000 !!!
lg1 db a-data ; syntax error – expression is not simple or relocatable

lg2 dw data-a; the assembler allows it but we obtain a linking error – "Error in file at 00129 – Invalid fixup recordname count…."

db a-$; = -5 = FB

c equ a-$; 0-6 = -6 = FA
d equ a-$; 0-6 = -6 = FA
e db a-$; 0-6 = -6 = FA
x dw x ; 07 20 !!!!!
x db x ; syntax error !
x1 dw x1 ; x1 = offset(x1) 09 00 at assembly time and 09 10 in the end…

db lg-a ; 04
db a-lg ; = -4 = FC
db [$-a] ; expression syntax error – a memory contents is not a constant computable at assembly time
db [lg-a] ; expression syntax error – a memory contents is not a constant computable at assembly time

lg1 EQU lg1 ; lg1 = 0 ! (BUG NASM !!!!) – it considers that an unitialized constant = 0
lg1 EQU lg1-a ; lg1 = 0 – WHY ????? It's a BUG !!! It works like that only because offset(a) = 0; if we put something else before a in data segment, offset(a) ≠ 0 and we will obtain a syntax error : "Recursive EQUs, macro abuse"

g34 dw c-2 ; -8

b dd a-start ; expression is not simple or relocatable !!! syntax error
(a – defined here, start – defined somewhere else)

dd start-a ; OK !!!!!!!! - (start – defined somewhere else, a – defined here)
POINTER DATA TYPE!!!!! (considered as FAR pointers subtraction !!!!)

dd start-start1 ; OK !!! – because they are 2 labels defined in the same segment! – Result will be a SCALAR DType !!!!

segment code use32

start:

```
mov ah, lg1    ; AH = 0
mov bh, c      ; BH = -6 = FA

mov ch, lg         ; OBJ format can handle only 16 or 32 byte relocation
mov ch, lg-a       ; CH = 04
mov ch, [lg-a]     ;  mov byte ptr DS:[4] – Access violation – most probably…

mov cx, lg-a        ;  CX = 4
mov cx, [lg-a]      ; mov WORD ptr DS:[4] – Access violation – most probably…

mov cx, $-a  ;  invalid operand type !!!!!  ($ – defined here, a – somewhere else)
mov cx, $$-a ; invalid operand type !! ; $$ from code segm and a from data segm !!

mov cx, a-$  ;  OK !!!!!! (a – defined somewhere else, $ – defined here)

mov ch, $-a  ;  invalid operand type !!!!!  ($ – defined here, a – somewhere else)
mov ch, a-$  ;  OBJ format can handle only 16 or 32 byte relocation
(a-$ is OK, but a-$ = POINTER !! – syntax error ! – because offset doesn't fit 1 byte !!)

mov cx, $-start  ; ok !!!
mov cx, start-$  ; ok !!! (both ok – the labels are defined in the same segment !!!)

mov ch, $-start  ; ok !!!  because RESULT is scalar ! (if it were pointer – syntax error !)
mov ch, start-$  ; ok !!!
```

mov cx, a-start ; ok !!! (a – defined somewhere else, start – defined here)
mov cx, start-a ; invalid operand type !!! (start – defined here, a – somewhere else)

start1:

mov ah, a+b ; NO syntax error !!!!!!!!!! BUT …. It is NO pointer arithmetic, NO pointers addition ; it is SCALAR addition - a+b = (a-$$) + (b-$$)

mov ax, b+a ; AX = (b-$$) + (a-$$) – SCALARS ADDITION !!!!
(somewhere else – here)

mov ax, [a+b] ; INVALID EFFECTIVE ADDRESS !!!! – THIS represents REALLY a pointers addition which is FORBIDDEN !!!!!!

var1 dd a+b ; syntax error ! - expression is not simple or relocatable
(so NASM doesn't allow "a+b" to appear in a DATA DEFINITION initialization, but ONLY as an INSTRUCTION OPERAND like above !!!

Conclusions:

Expressions of type et1 – et2 (where et1 and et2 are labels – either code or data) are syntactically accepted by NASM,

- Either if both of them are defined in the same segment

- Either if et1 belongs to a different segment from the one in which the expression appears and et2 is defined in this latter one. In such a case, the expression is accepted and the data type associated to the expression et1-et2 is POINTER and NOT SCALAR (numeric constant) as in the case of an expression composed of labels belonging to the same segment. (So SOMEWHERE ELSE – HERE OK !, but HERE – SOMEWHERE ELSE NO !!!)

  Subtracting offsets specified relative to the same segment = SCALAR
  Subtracting pointers belonging to different segments = POINTER

The name of a segment is associated with “Segment’s address in memory at run-time”, but this value isn’t accessible to us, this being decided only at loading-time by the OS loader. That is why, if we try to use the name of a segment explicitly in our programs we will get either a syntax error (in situations like $/a-data – “HERE – SOMEWHERE ELSE”) either a linking error (in situations like data-a - SOMEWHERE ELSE – HERE) because practically the name of a segment is associated with its FAR address (which will be known only at loading time, so it will be available only at run-time; we notice though that segment_address is considered to be “SOMEWHERE ELSE”) and it is NOT associated to "The Offset of a segment is a constant determinable at assembly time" (like it is in 16 bits programming for example, where segment_name at assembly time = its offset = 0). In 32 bits programming, the offset of a segment is NOT a constant computable at assembly time and that is why we cannot use it in pointer arithmetic expressions !

Mov eax, data ; Segment selector relocations are not supported in PE files (relocation error occurred)

You can check for instance that NASM and OllyDbg always provide for start offset (DATA) = 00401000, and offset(CODE) = 00402000. So here the offsets aren’t zero as in 16 bits programming !! So from this point of view there is a big difference between VARIABLES NAMES (which offsets can be determined at assembly time) and SEGMENTS NAMES (which offsets can NOT be determined at assembly time as a constant, this value being known exactly like the segment’s address ONLY at loading time).

If we have multiple pointer operands, the assembler will try to fit the expression into a valid combination of pointer arithmetic operations:

Mov bx, [(v3-v2) ± (v1-v)] – OK !!! (adding and subtracting immediate values is correct !!)

If not, we will obtain “Invalid effective address ”syntax error”:

Mov bx, [v2-v1-v] ; Invalid effective address !

Mov bx, v2-v1-v ; Invalid operand type ! mov bx, v2-(v1+v) – so we conclude that v1+v is accepted as a SCALAR in interpretation a+b = (a-$$) + (b-$$), ONLY IF a+b appears AS

ITS OWN or besides other SCALAR expressions ! but NOT in combination with POINTER expressions !!

Mov bx, v2+v1-v ; ok = v2+scalar
Mov bx, [v3-v2-v1-v] ; Invalid effective address !
Mov bx, v3-v2-v1-v ; OK !!! – because... Mov bx, v3-v2-v1-v = Mov bx, (v3-v2)-(v1+v) = subtracting immediate values (SCALARS !!!)

The direct vs. indirect addressing variant is more permissive as arithmetic operations in NASM (due to the acceptance of “a + b” type expressions) but this does not mean that it is more permissive IN POINTER OPERATIONS !!!

Mov bx, (v3±v2) ± (v1±v) – OK !!! (adding and subtracting immediate values is correct!!)

The expressions in parantheses (in both yellow and green variants) MUST BE SCALARS !!!!!!!!

Varianta in limba romana a textului de dupa concluzii

Numele unui segment este asociat cu "Adresa segmentului in memorie în faza de execuţie" însă aceasta nu ne este disponibilă/accesibilă, fiind decisă la momentul încărcării programului pt execuţie de către încărcătorul de programe al SO. De aceea, dacă folosim nume de segmente în expresii din program în mod explicit vom obţine fie eroare de sintaxă (în situaţii de tipul $/a-data - “AICI – ALTUNDEVA”) fie de link-editare (în situaţii de tipul data-a - ALTUNDEVA – AICI), deoarece practic numele unui segment este asociat cu adresa FAR al acestuia (adresă care nu va fi cunoscută decât la momentul încărcării programului, deci va fi disponibilă doar la run-time; observăm astfel că adresă_segment este considerată ca “ALTUNDEVA”) şi NU este asociat cu "Offset-ul segmentului în faza de asamblare, fiind o constanta determinata la momentul asamblarii" (aşa cum este în programarea sub 16 biţi de exemplu, unde nume_segment în faza de asamblare = offset-ul său = 0). Ca urmare, se constată că în programarea sub 32 de biţi numele de segmente NU pot fi folosite in expresii !!

Mov eax, data ; Segment selector relocations are not supported in PE files (relocation error occurred)

Sub 32 biţi offset (data) = vezi OllyDbg – Intotdeauna DATA segment incepe la offset-ul 00401000, iar CODE segment la offset 00402000. Deci offset-urile începuturilor de segment nu sunt 0 la fel ca şi la programarea sub 16 biţi. Din acest punct de vedere sub 32 biţi este o mare diferenţă între numele de variabile (al căror offset poate fi determinat la asamblare) şi numele de segmente (al căror offset NU poate fi determinat la momentul asamblării ca şi o constantă, această valoare fiind cunoscută la fel ca şi adresa de segment doar la momentul încărcării programului pt execuţie - loading time).

Daca avem mai mulţi operanzi pointeri, asamblorul va încerca să încadreze expresia într-o combinaţie validă de operaţii cu pointeri:

Mov bx, [(v3-v2) ± (v1-v)] – OK !!! (adunarea şi scăderea de valori imediate este corectă !!)

…însă nu trebuie să uităm că fiind vorba despre ADRESARE INDIRECTA rezultatul final al expresiei cu operanzi pointeri din paranteze trebuie să furnizeze în final UN POINTER !!!... din această cauză în cazul adresării indirecte nu vor fi niciodată acceptate expresii cu operanzi adrese de forma “a+b”

Dacă expresia nu va putea fi încadrată într-o combinaţie validă de operaţii cu pointeri, vom obţine eroarea de sintaxă “Invalid effective address”:

Mov bx, [v2-v1-v] ; Invalid effective address !

Mov bx, v2-v1-v ; Invalid operand type !

Mov bx, v2-v1-v ; Invalid operand type ! mov bx, v2-(v1+v) – de unde concluzionăm că v1+v este acceptată ca şi SCALAR în interpretarea a+b = (a-$$) + (b-$$), doar dacă apare DE SINE STATATOR sau în combinaţie cu alţi SCALARI , dar NU şi nu în combinaţie cu expresii de tip POINTER !!!

Mov bx, v2+v1-v ; ok = v2+scalar

Mov bx, [v3-v2-v1-v] ; Invalid effective address !

Mov bx, v3-v2-v1-v ; OK !!! – deoarece ?... Mov bx, v3-v2-v1-v = Mov bx, (v3-v2)-(v1+v) = scădere de valori imediate (SCALARE!)

Varianta de adresare directă vs.indirectă este mai permisivă ca operaţii aritmetice în NASM (din cauza acceptării expresiilor de tip “a+b”) însă asta nu înseamnă că este mai permisivă IN OPERATIILE CU POINTERI !!!

Mov bx, (v3±v2) ± (v1±v) – OK !!! (adunarea şi scăderea de valori imediate este corectă !!)