**Reguli generale**:

Primul pas este sa convertim fiecare numar indifferent in ce baza e, in baza 16. Daca numarul este negative, il vom transforma in nr pozitiv astfe; daca x<0 => x=256+x (daca e octet, aici de fapt e o scadere cu complementul fata de 2)

Al doilea pas ar fi sa adaugam 0-uri nesemnificative pana obtinem exact numarul de cifre hexa pe care il dorim:

Pentru DB 2 cifre hexa, DW 4 cifre hexa, DD 8 cifre hexa, DQ 16 cifre hexa

Apoi, grupam fiecare dintre numerele obtinute cate 2, si se salveaza in memorie in ordine inversa (little-endian)

Pentru numere negative:

* Daca x<0, prima data transformam |x| in baza 2 (cu atatia de 0 nesemnificativi cati trebuie pentru a umple toti bitii)
  + 8 biti in total pentru byte
  + 16 biti in total pentru word
  + 32 biti in total pentru dword
  + 64 biti in total pentru qword
* Apoi inversam toti bitii si adaugam 1 (una dintre reguli)
* Transformam nr in baza 16, si apoi se salveaza in memorie conform little-endian

Cazul de numere care nu intra in intervalul de reprezentare:

* **BAZA** **2**:
  + Vom considera doar ultimii 8 biti din numar (pentru byte) sau ultimii 16 biti (pentru word) etc
  + Se transforma ultimii 8/16/32/64 de biti in baza 16, si se salveaza in memorie folosind little-endian
* **BAZA 16**:
  + Transformam nr in baza 2, apoi se aplica aceleasi reguli ca mai sus
  + Putem lua direct ultimele 2 cifre hexa (byte sau 4 pt word sau 8 pentru dword etc) si acestea se salveaza in memorie
* **NUMERE** **POZITIVE**:
  + Transformam nr in baza 2, apoi se aplica aceleasi reguli ca mai sus (sau 16 si luam ultimele cifre hexa)
    - Alternativ: Daca numarul e mic, putem pur si simplu sa scadem un 256, si daca rezultatul e un nr din intervalul de reprezentare, atunci se trateaza in mod normal, numarul salvat in memorie fiind nr nou obtinut
  + Exemple care ar putea sa vina:
    - A db 260 ⬄ A db 260-256
    - A db 300 ⬄ A db 300-256
* **NUMERE NEGATIVE**:
  + Transformam nr in modul in baza 2
  + Inversam toti bitii si adaugam un 1
  + Se aplica reguli ca in baza 2
    - Alternativ: Daca numarul e mare, atunci adunam un 256 la el, si daca rezultatul este in in intervalul de reprezentare cu semn atunci:
      * Daca nr e pozitiv, atunci se converteste in baza 16, si salveaza in memorie ca atare
      * Daca nr e negative, se aplica aceleasi reguli ca la nr negative (metoda explicate sus)
  + Exemple care ar putea sa vina:
    - A db -129 ⬄ A db -129+256 (care apartine intervalului de reprezentare pe byte
    - A db -150 ⬄ A db -150+256

**SIRURI**:

Daca un sir e definit cu DB:

A db ‘abcdefgh’

* Se salveaza in memorie asa cum apar

Daca un sir e definit cu DW:

A dw ‘abc’ ⬄ a dw ‘ab’,’c’ ⬄ a db ‘abc’,0 ⬄ a db ‘a’,’b’,’c’,0

* |’a’ ‘b’ ‘c’ 0|

A dw ‘a’,’b’,’c’ ⬄ a db ‘a’,0,’b’,0,’c’,0

* |’a’ 0 ‘b’ 0 ‘c’ 0|

Daca un sir e definit cu DD:

A dd ‘abcdef’ ⬄ a dd ‘abcd’,’ef’ ⬄ a db ‘abcdef’,0,0

* |’a’ ‘b’ ‘c’ ‘d’ ‘e’ ‘f’ 0 0|

A dd ‘ab’,’cd’ ⬄ a db ‘ab’,0,0,’cd’,0,0

* |’a’ ‘b’ 0 0 ‘c’ ‘d’ 0 0|

**Concluzii**:

Daca nu avem sirul de caractere definit cu doubleword NASM incearca sa combine cat mai multe carctere intr-unul pana la maxim 4, si minim 1. “,” Desparte un doubleword/word de altul, fortand astfel sa se adauge atati de 0 cati sunt necesari pentru ca lungimea sirului de dinainte de ‘,’ si intre cealalta ‘,’ sa se imparta exact la 4/2 (dword/word).

A dw ‘abc’,’de’; Se adauga un 0 intre ‘abc’ si’de’

A dd ‘abc’,’de’; Se adauga un 0 intre ‘abc’ si ‘de’ si doi de 0 la final (dupa ‘de’)

La DQ aceleasi reguli, doar ca impartirea o sa fie la 8

**Resb/Resw/Resd/Resq [n]**

Resb/Resw/Resd/Resq initializeaza cu 0 n byte/word/dword/qword

A resb 2

B db -1

* |00 00 FF|

A resw 2

B db -1

* |00 00 00 00 FF|

**Observatii**:

In segmentul de data putem avea db/dw/dd/dq/resb/resw/resd/resq fara o eticheta si nu da eroare!!!!

Resb 2

A db -1

* |00 00 FF|

Prin ‘c’ m-am referit la codul ascii al caracterului c (2 cifre hexazecimale) -> un byte (O constanta)

* EQU nu creaza un spatiu de memorie!!!!
* Pot aparea scaderi de caractere, care practice este scaderea dintre codul lor ASCII

“**$**” -> In segmentul de date reprezinta adresa actuala in segmentul de date (mai usor de imaginat ar fi adresa unde se “adauga” elemente in memorie in continuare)

A db $-A; A=0

A db 1,$-A;A=1,a

Secventa:

A db 1,2,3,4

B equ $-A; B=4 Deoarece $-A reprezinta nr de octeti dintre adresa unde se adauga in memorie date, si adresa lui A, deci reprezinta nr de octeti dintre locul “current” si variabila A

C dw 1,2,3,4

D equ $-C; C=8 Deoarece intre locul “current” si C sunt 4 word-uri deci 8 octeti

E equ $-A; E=12 Deoarece intre locul “current” si A sunt 4 word-uri si 4 octeti, deci 12 octeti

* Se observa ca etichetele definite cu EQU nu modifica valoarea lui $
* $ creste cu 1 de fiecare data cand avem definit o eticheta cu db, cu 2 daca e definita cu dw, cu 4 daca e definite cu dd, si cu 8 daca e definita cu dq
* Daca etichetele sunt definite cu resb n, atunci la valoarea lui $ va creste cu n, daca e definita cu resw n, creste cu n\*2 si asa mai departe
* Observam ca prima data adauga numarul in memorie, si apoi creste conform acestor reguli valoarea lui $

$$ de multe ori are exact aceasi valoare cu $ (se refera la segmentul activ), dar segmentul activ e segmentul actual daca lucram ne-multimodul