Limbaj masina = totlitatea instructiuniilor masina puse la dispozitie de processor

Limbaj de asamblare = Limbaj symbolic (instructiuniile coincide cu instructiuniile masina)

Eticheta = Nume de variabila sau zona de memorie

- ➔ 2 categorii de etichete: **eticheta cod**: apar in cadrul secventelor de instructiuni pt a define destinatia de dransfer de date

  eticheta de date: unele locatii de memorie

- ➔ **Valoarea** unei etichete in limbaj de asamblare este un nr intreg reprezentand adresa (FAR) a instructiunii, directive sau datelor ce urmeaza etichetei
- ➔ Daca numele etichetei este pusa intre paranteze drepte: [] inseamna ca ne referim la valoarea din memorie a ei

Instructiuni = Mnemonice ce sugereaza actiunea

- ➔ Asamblorul genereaza octetii care genereaza respectiva instructiune
- ➔ Mnemonice de instructiuni si nume de directive

Directive = Indicatii date asamblorului in scopul generarii corecte de octeti

- ➔ Tipuri de directive:
- ➔ **DIRECTIVA SEGMENT** nume [tip] [ALIGN=aliniere] [combinare] [utilizare] [CLASS=clasa]
  - Numele segmentului = adresa de segment
  - TIP: permite selectarea unui model de folosire al segmentului si poate fi:
    - code – segmentul va contine cod: poate fi citi si executat, dar nu scris
    - data – segmentul va contine date: poate fi citit si scris, dar nu executat
    - rdata – segmentul contine doar constante: poate fi doar citit
  - ALINIERE: specifica multiplul numarului de octeti la care trebuie sa inceapa segmentul respectiv: putere a lui 2 intre 1 si 4096: este 1 implicit
  - COMBINARE: modul in care se combina cu alte module:
    - PUBLIC: e vizibil de orice alt modul, poate fi combinat cu orice alt modul
    - COMMON: acest modul se suprapune peste inceputul oricarui alt modul cu acelasi nume => un segment cu dimensiunea egala a celui mai mare segment cu acelasi nume
    - PRIVATE: nu e vizibil, nu poate fi combinat cu nici un alt modul
    - STACK: segmentele cu acelasi nume vor fi concatenate
  - CLASS: are rolul de a permite stabilirea ordinii in care editorul de legaturi plaseaza segmentele in memorie. Toate segmentele cu aceeasi clasa vor plasate intr-un singur bloc
  - segment code use32 class=CODE
  - segment data use32 class=DATA
- ➔ **DIRECTIVE PENTRU DEFINIRE A DATELOR**:
  - Definire date = declarare (specificare atribute) + alocare (rezervare spatiu de memorie)
  - Tip de date (TD) = byte/word etc (Dimensiune de reprezentare)
  - Variabila = nume, set de atribute, adresa, valoare
  - Set de atribute = TD, Scope (domeniu de vizibilitate), durata de viata, clasa de memorie, etc (in C)
  - Directive de definire: DB, DW, DD, DQ, DT (10 octeti)
    - A db 1,2,3,4; Se intializeaza patru variabile in memorie (4 octeti) sau un tablou de 4 elemente
- ➔ **DIRECTIVE DE DATE NEINITIALIZATE** (valoarea initializarii e 0):
  - RESB, RESW, RESD, RESQ, REST
- ➔ **DIRECTIVA TIMES**:
  - Permite asamblarea repetata a unei instructiuni sau definitii de date:
  - TIMES factor expresie; factor e un numar, si expresie se va copia de factor (in asamblare)
- ➔ **DIRECTIVA EQU**:

- Permite atribuirea unei valori numerice constante sau sir de caractere unei etichete fara alocarea de spatiu de memorie sau generare de octeti
- Nume EQU expresie

Contor de locatii = Numar intreg generat de asamblor

➔ În fiecare moment, valoarea contorului coincide cu numărul de octeţi generaţi corespunzător instrucţiunilor şi directivelor deja întâlnite în cadrul segmentului respectiv

Operanzii = parametrii care definesc valoriile ce vor fi prelucrate de instructiuni sau directive

➔ Pot fi: registrii, constante, etichete, expresii, cuvinte cheie sau alte simboluri
➔ 3 tipuri:
- Operanzi imediati: 1, 2, 3 etc
  - Constante numerice determinabile la momentul asamblarii
  - 0x 0h pentru baza 16, 0d 0t pentru zecimal, 0o 0q pentru octal, ob oy pentru binar
  - Aceste litere poti fi puse si la final: 101010b, 1234AFh
  - Deplasamentele etichetelor sunt variabile determinabiile la momentul asamblarii
- Operanzi registru: eax, ebx, esi etc
  - Mov eax,ebx -> accesare directa
  - Mov eax,[ebx] -> accesare indirecta
- Operanzi in memorie: byte[a], word[b] etc
  - Operandul cu adresare directa: o constanta care indica adresa (segment si deplasament) pot fi: etichete sau valoarea contorului de locatii (b db $-a)
    - Deplasamentul e calculat in timpul asamblarii, adresa fiecarui operand este calculata in momentul editarii de legaturi (LINKING TIME), iar adresa fizica efectiva in momentul incarcarii
    - Un deplasament este intotdeauna raportat la un segment (CS, SS, DS etc)
    - Asocieri implicite de asociere:
      - CS daca pentru etichete de cod destimatie (jmp repeta, call [printf], ret, jz etc)
      - SS daca in SIB se foloseste ESP sau EBP ca baza
      - DS pentru restul
  - Operandul cu adresare indirecta: Utilizeaza registrii pentru a indica zone de memorie:
    - [baza] + [index*scala] + [constanta]
    - Constanta este expresie a carei valoare este determinabila la momentul asamblarii
    - Pot fi rescrise si formula sa fie tot valida: mov al,[ebx*3]; ⇔ mov al,[ebx+ebx*2]
    - Scala poate fi 1 2 sau 4

Operatori = pentru combinarea, compararea, modificarea si analiza operatoriilor.

TIPURI DE OPERATORI:

➔ - (scadere), + (adunare), ~(complement fata de 1, negare bit cu bit), !(negare logica !0=1 si !a=0 daca a!=0), *(inmultire), / (catul impartirii fara semn), // (catul impartirii cu semn), % (restul impartirii cu semn), %% (restul impartirii fara semn), << (deplasare la stanga pe biti), >> (deplasare la dreapta pe biti) & („si” bit cu bit), | („sau” bit cu bit), ^ („sau exclusiv” bit cu bit)
➔ Operator de **indexare**: „[]” cand e folosit cu formula de calcul al offset-ului
➔ Operator de **specificare al segmentului**: „:” ex: mov eax,[SS:EBX+4] (se refera exact la segmentul de stiva si suprascrie regulile implicite de setare al segmentului)
➔ Operator de **tip**: byte, word, dword, qword -> forteaza o expresie sa fie tratata ca avand o anumita dimensiune (se foloseste adesea cand ne raportam la memorie)
➔ Valoarea operanziilor este determinabila la momentul asamblarii pentru operanzii imediati si operanzii cu adresare directa (offset-ul doar), in momentul incarcarii (LOADING TIME) pentru operanzii cu adresare directa (offset + segment -> adresa FAR) (acest pas implica un proces numit RELOCARE a adreselor

adica determinarea adreselor de segment pe baza selectorului de segment) si in momentul executiei (RUN-TIME) pentru operanzii registru si cei adresati indirect (cu formula de calcul al offset-ului in care intra o formula bazata si/sau indexat-scalata)

Expresie = Operanzi + Operatori

- ➔ Operatorii sunt determinabili la momentul asamblarii (cu exceptia formulii de la 2 noaptea)

Depasire = O conditie/situatie matematica care exprima faptul ca rezultatul unei operatii nu a incaput in spatiul rezervat acestuia

- ➔ Prin setarea CF=1 sau OF=1, procesorul ne transmite mesajul ca ambele interpretari in baza 10 (cu semn sau fara semn) sunt INCORECTE (CF=1 => interpretarea fara semn are rezultatul gresit; OF=1 => interpretarea cu semn are rezultatul gresit)

Adresa unei locatii = Nr de octeti consecutivi dintre inceputul memoriei RAM si inceputul locatiei respective

Segment = O succesiune continua de locatii de memorie, menita sa deserveasca scopuri similare

- ➔ O diviziune logica a memoriei unui program
- ➔ Contine o adresa de baza (inceput), limita (dimensiune), si tipul acesteia (toate reprezentate pe 32 de biti)
  - Segment de cod: contine instructiuni masina (CS – selector de segment)
  - Segment de date: contine definiri de date (DS-selector de segment)
  - Segment de stiva: SS-selector de segment
  - Extra segment: ES
  - FS, GS -> selectori care indica spre segmente suplimentare neavand atribuiri implicite
  - EIP -> offsetul instructiunii curente in cadrul segmentului de cod curent

Specificare de adresa = O pereche: Selector de segment si offset ([DS:EBX])

- ➔ Selectorul de segment este definit de catre sistemul de operare (SO)
- ➔ Acea pereche se mai numeste ADRESA FAR

Adresa NEAR = atunci cand o adresa se precizeaza doar prin offset

- ➔ Daca se depaseste limita => MEMORY VIOLATION ERROR
- ➔ Calculul pentru validare este facuta de componenta ADR din BIU
- ➔ Este intotdeauna in interiorul din cele 4 segmente active
- ➔ Daca segmentele incep de la 0 => Model de memorie flat

Paginare = Implica impartirea memoriei virtuale in pagini care sunt asociate memoriei fizice disponibile

- ➔ Sarcina Sistemului de operare (SO)

Instructiune masina = O secventa de la 1 pana la 15 octeti

- ➔ Reprezinta o operatie de efectuat
- ➔ Maxim 2 operanzi (sursa si destinatie)
- ➔ Maxim un operand din memoria RAM
- ➔ Forma generala:
  - [prefixe] + cod + [Mod R/M] + [SIB] + [deplasament] +[imediat]
  - Prefixe: controleaza modul in care o instructiune se executa
  - Mod R/M: Natura si locul operanziilor (registru/memorie) si se poate combina cu octetul SIB
  - Deplasament (displacement): forma de adresare particulara
  - Imediat: consanta numerica

Mod adresare = Modul in care ne adresam la un operand

- ➔ Mod adresare la memorie:

- Adresare directa: daca apare doar o constanta (imediat si o eticheta): mov **byte[a]**,1
- Adresare bazata: daca apare baza: mov **byte[ebx]**,1
- Adresare scalat-indexata: daca apare index (implicit si scala): mov **byte[ebx*2]**,1
- Adresa relativa: pozitia urmatoarei instructiuni de executat in raport cu pozitia curenta
- O adresa care nu este directa se numeste indirecta

➔ Mod registru: mov **EAX**,17
➔ Mod imediat: mov EAX,**17**

Arhitectura microprocesoarelor:

➔ **EU** = Executive unit
- Executa instructiuniile masina prin intermediul componentei ALU (aritmetic and logic unit)

➔ **BIU** = Bus interface unit
- Pregateste executia fiecarei instructiuni masina

➔ Cele doua lucreaza in paralel

Flag = un indicator reprezentat pe un bit (0 sau 1)

Conversie distructive = Schimba efectiv numarul

➔ CBW, CWD etc
➔ Movzx, movsx, mov

Conversie nedistructiva = Nu schimba efectiv numarul

➔ BYTE, WORD, DWORD, QWORD

Resurse la subrutine (programare Multimodul)

➔ **Resurse volatile** = Acei registrii pe care conventia de apel ii defineste ca apartinand subrutinei apelate, ca atare apelantul avand datoria sa le salveze daca are nevoie de ele, si sa le restaureze dupa (adica registrii folositi in alt modul ca parametrii)
➔ **Resurse nevolatile** = Adrese de memorie/registrii care nu ii apartin explicit subrutinei chemate, iar ca atare, daca acesta doreste sa efectueze modificari este necesar sa le salveze inainte de iesire
➔ Diferenta apare in privinta cui are obligatia sa faca salvarea si returnarea
➔ Volatile -> Apelantul (unde apelam o functie)
➔ Nevolatile -> Codul apelant (in subprogram)

Flaguri

Scurtă definiție

CF -> carry flag: valoarea transportului final intr-o adunare sau scădere

OF -> overflow flag: valoarea 1 daca doua numere cu acelasi semn, dau unul de semn opus

ZF -> zero flag: 1 daca rezultatul ultimei operatii effectuate este 0

SF -> sign flag: Ia valoarea bitului de semn din rezultatul ultimei operatii effectuate

DF -> direction flag: 0 – deplasarea in sir se face de la stanga la dreapta, daca e 1 invers

PF -> parity flag: ultimul bit (nu stiu exact)

AF -> auxialiry flag: valoarea transportului de la bitul 3 la bitul 4

TF -> trap flag: val 1 daca masina se opreste dupa fiecare instructiune -> nu poate fi schimbat

IF -> interruption flag: pentru sectiuni critice -> nu poate fi schimbat

Instrucțiuni

CLC/STC/CMC

Syntaxa: **CLC/STC**

Efect: pune in CF valoarea 0/1/!CF

Restricții:

Pe scurt: CF=0/1/!CF

CLD/STD

Syntaxa: **CLD/STD**

Efect: pune in DF valoarea 0/1

Restricții:

Pe scurt: CF=0/1

Speciale

PUSHF/POPF

Syntaxa: **PUSHF/POPF**

Efect: pune/scoate toate flag-urile pe stiva

Restricții:

Pe scurt: EFLAGS=[ESP] / [ESP] = EFLAGS

MODURI DE SETARE

CARRY FLAG

CF se seteaza la 1 daca avem o adunare/sau o scadere care nu incape pe acelasi interval de reprezentare

Interval de reprezentare pe octet: [0,255]

- ➔ Functioneaza aceasta regula doar daca folosim numerele reprezentate fara semn (Daca avem un nr negativ de ex: -1, -1 ca numar pozitiv reprezentat pe octet va fi 256 (2^8) – 1),
  Deci vom considera -1 = +255 (se mai numesc si valori complementare)

Regula generala: daca x<0 => x=256+x (Aici e de fapt o adunare cu un nr negativ)

!! Daca rezultatul e mai mic ca 0 sau mai mare decat 255 atunci CF=1, si in rest CF=0

Mare atentie daca avem adunare sau scadere. La adunare vom considera numerele negative ca fiind 256 – acel numar (daca e reprezentat fara semn), dar la scadere daca scadem un numar pozitiv din alt nr pozitiv atunci lasam numarul asa cum este

Exemple:

ADUNARE:

1)

```
mov al,100
mov ah,200
add al, ah
```

- ➔ La sfarsitul secventei CF=1 deoarece 100+200=300 > 255

2)

```
mov al,100
mov ah,-1
add al, ah
```

- ➔ La sfarsitul secventei CF=1 deoarece 100+(-1) = 100 + (256-1) = 100 + 255 = 355 > 255

SCADERE:

1)

```
mov al,100
mov ah,101
sub al, ah
```

- ➔ La sfarsitul secventei CF=1 deoarece 100-101=-1 < 0

2)

```
mov al,100
mov ah,-1
sub al, ah
```

- ➔ La sfarsitul secventei CF=1 deoarece 100-(-1)=100-(256-1)=100-255=-155 < 0

Concluzie: CF va fi setat daca rezultatul ultimei operatii efectuate nu este in intervalul [0,255]

->Toate numerele negative (din definitie) le transformam in 256-nr respectiv

->NU se aplica regula a+(-b)=a-b !!!!!!!!!!! (-b nr negativ din definitie, si il vom transforma ca si in [0,255]

->Ca regula principala, consideram ca nu exista numere negative, si pe toate le transformam in pozitive cu regula de mai sus

OVERFLOW FLAG

Interval de reprezentare pe octet: [-128,127]

Spre deosebire de ce faceam inainte, acum ca sa vedem usor cum se seteaza OF, transformam numerele pozitive din intervalul (127,255] in numere negative astfel: 128 = 128-256 = -128; 200 = 200-256=-56

Regula generala: daca x>127 => x=x-256 !!!!!

Daca rezultatul este mai mare ca 127 sau mai mic ca -128 atunci OF=1, si OF=0 altfel

Exemple:

ADUNARE:

1)

```
mov al,100
mov ah,100
add al, ah
```

➔ OF-1 deparece 100+100 = 200 > 127

2)

```
mov al,-100
 mov ah,156
add al, ah
```

➔ OF-1 deparece -100+156 = -100 + (156-256) = -100 + -100 = -200 < -128

SCADERE:

1)

```
mov al,100
mov ah,-100
sub al, ah
```

➔ OF=1 deoarece 100-(-100) =100+100=200>127

2)

```
mov al,100
mov ah,156
sub al,ah
```

➔ OF=1 deoarece 100-(156) =100-(156-256)=100-(-100)=200>127

Concluzii:

Daca numarul este mai mare decat 127, el trebuie convertit cu regula precizata ca sa aflam practic daca OF=1

Daca avem o scadere cu un numar negativ din interval (-128<=x<0) atunci functioneaza regula a-(-b)=a+b !!

OBSERVATII CF SI OF

Pentru imul si mul CF=OF=1 daca rezultatul obtinut nu este in intervalul de reprezentare stabilit (numere pozitive pentru mul si numre negative si pozitive pentru imul)

Regulile de transformare raman aceleasi pentru numere din intervalul [0,255] (x<0 => x=256+x) sau [-128,127] (x>127 => x=x-256)

Exemple:

1)

```
mov al,-1
mov ah,2
mul al
```

➔ CF=OF=1 deoarece -1 * 2 = (256-1)*2 = 255*2>255 (aici -1 nu e in intervalul de reprezentare, deci l-am transformat)

2)

```
mov al,-1
mov ah,2
imul al
```

➔ CF=OF=0 deoarece -1*2=-2 (Aici -1 si -2 sunt in intervalul de reprezentare)

3)

```
mov al,156
 mov ah,3
```

imul al

➔ CF=OF=1 deoarece 156*3=(156-256)*3=-100*3=-300<-128 (aici 156 nu e in intervalul de reprezentare)

Aici am considerat toate exemplele reprezentate pe octet, dar tot aceleasi reguli se aplica si pe word si dword

La impartire daca rezultatul final nu incape pe intervalul de reprezentare NASM produce „run-time-error" (programul „crapa") deci nu se face analiza lui CF si OF in acele cazuri

ZERO FLAG

ZF=1 daca rezultatul ultimei operatii efectuate

Exemple nasoale:

1)

```
mov al,15
mov ah,241
add al, ah
```

➔ ZF=1 deoarece 241+15=256 = 256-256 = 0 (256 nu incape pe un byte, deci se salveaza doar ultimii 8 biti din rezultat si CF=1)

2)

```
mov al,-1
mov ah,255
sub al, ah
```

➔ ZF=1 deoarece -1-255 = (256-1)-255 = 255 – 255 = 0 (aici am putea converti si invers)

➔ ZF=1 deoarece -1-255 = -1-(255-256)=-1-(-1)=-1+1=0

Observatii:

Cand facem adunarea sau scaderea pentru a analiza ZF, trebuie neapart sa avem numere din acelasi interval de reprezentare, fie cu numere pozitive fie cu numere negative (si pozitive) (rezultatul e tot timpul corect daca convertim la acelasi interval de reprezentare fie pe primul fie pe al doilea)

SIGN FLAG

Bitul de semn al ultimei operatii efectuate

Pentru adunare si scadere:

Convertim amandoua numerele la numere fara semn cu regula de mai sus (daca x<0 =>x=256+x)

Adunam normal numerele.

Daca rezultatul este in depasire (CF=1) atunci rez=rez-256

Daca noul rezultat este mai mic decat 128 atunci SF=0, altfel SF=1

Exemple:

1)

```
mov al,10
mov ah,-1
add al, ah
```

➔ SF=0 deoarece 10 + (-1) = 10+(256-1)=10+255=265 (depasire)=265-256 = 9 <128

2)

```
mov al,-1
mov ah,-1
add al, ah
```

➔ SF=1 deoarece (-1)+(-1)=255+255 (depasire)=255+255-256 = 254 >=128

Pentru inmultire si impartire:

La imul daca inmultim un nr pozitiv cu un numar negativ atunci SF=1 si SF=0 altfel

La mul -> val bitului de semn din al/ax/dx (din observatii ca nu am gasit in curs)
La div si idiv -> Nu am vazut schimbari IDK

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
- ➔ **NUMERE POZITIVE**:
  - Transformam nr in baza 2, apoi se aplica aceleasi reguli ca mai sus
    - 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,$-1;A=1

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

Operatorii efectueaza calculea cu valori constante

OPERATORI LOGICI PE BITI

“~” -> Neaga fiecare bit din numarul de dupa el

~a=1 daca a = 0 si 0 daca a=1

Exemple:

a db ~10101010b; |55| 5h=0101b a=55h

b db ~10000000b; |7F| 7h=0111b Fh=1111b b=7Fh

“&” ->face “si” bit cu bit dintre cele doua constante

Exemple:

a db 11111111b&0b; |00| a=00h

b db 11111111b&11110000b; |F0| Fh=1111b b=F0h

“|” ->face “sau” bit cu bit dintre cele doua constante

Exemple:

a db 11111111b|0b; |FF| a=FFh

b db 00001111b|11110000b; |FF| a=FFh

“^” ->face “sau exclusiv” bit cu bit dintre cele doua constante

Bitul resultant va avea valoarea 1 daca cei doi biti au valori diferite (unul 1 si celalalt 0)

Exemple:

a db 11110000b^11111111b; |F0| a=F0h

b db 11111111b^11111111b; |00| a=00h

“!”

Neaga o expresie intreaga

!a=1 daca a=0, sau 1 altfel

Exemple:

a db !10101100b; |00| a=00h

b db !0b; |01| a=01h

OPERATORI DE DEPLASARE PE BITI

“<<” -> Deplaseaza la stanga constant data ca parametru

->Valoriile ramase se completeaza cu 0

Exemple:

a db 11111111b<<4; |F0| a=11110000b

a db 10000000b<<1; |00| a=00000000b

Consideram urmatoarea secventa:

a db 0FFh

b db 0b<<8

Rezultatul final din memory layout va fi |FF|00|, deci nu se deplaseaza in stanga memoriei

Dar, consideram urmatoarea secventa:

mov al,11111111b<<8

In AL va fi 0, si in AH va fi FF

->Deci daca folosim Deplasariile pe biti impreuna cu registrii, se vor deplasa efectiv bitii mai departe

“>>” -> Deplaseaza la dreapta constant data ca parametru

->Valoriile ramase se completeaza cu 0

Exemple:

a db 11111111b>>4; |0F| a=00001111b

a db 00001111b>>1; |07| a=00000111b

Consideram urmatoarea secventa:

a db 0b>>8

b db 0FFh

Rezultatul final in memory layout va fi |00|FF|, deci nu se deplaseaza in dreapta memoriei

Dar, consideram urmatoarea secventa:

Mov eax,-1

Mov ah,11111111b>>8

->In AH vom avea 0, si in al 11111111b

->Ca si concluzii ar fi ca in cazul registriilor nu se “trag” valoriile din stanga sau dreapta, daca incepem cu ah, nu se vor lua biti din partea high din eax, ci se va complete cu 0, si se vor deplasa toti bitii de dupa ah

RULARE

Pentru rularea unui program multimodul:

Pas 1: Adaugare in program ALINK.exe si NASM.exe (in nasm din asmtools)

Pas 2: Rezolvare probleme (scrii cod)

Pas 3: Pozitionare in director in Command Prompt: cd __PATH__

Pas 4: **Comenzi**:

nasm -f obj main.asm

nasm -f obj second.asm

alink main.obj second.obj -oPE -subsys console -entry start

main.exe

- ➔ main e numele fisierului principal si second numele fisierului secundar

REGULI

- ➔ Sa nu fie numita a doua functi ca si numele fisierului (da eroare la link-editare)
- ➔ Cand apelam functia sa se scrie fara paranteze drepte []: call functie nu call [functie]
- ➔ Cand apelam functii normale se scriu intre paranteze drepte []

CUM SE SCRIE DE LA 0:

- ➔ In modulul principal trebuie declarant in **SEGMENTUL DE DATE** eticheta de unde porneste al doilea modul
- ➔ In modulul secundar trebuie declarata in **SEGMENTUL DE DATE** eticheta de unde porneste modulul ca fiind globala

TEORIE MULTIMODUL

Apelul unei procedure se face cu ajutorul instructiunii CALL, care poate fi apel direct sau indirect

CALL operand

- ➔ Transfera controlul la adresa desemnata de operad, dar si salveaza in stiva adreasa urmatoarei instructiuni dupa CALL
- ➔ Echivalent cu:
  - o Push Urmatoare
  - o Jmp operand
  - o Urmatoare:

RET [n]

- ➔ Terminarea executiei secventei apelate este marcata de intalnirea unei instructiuni RET
- ➔ Acesta preia din stiva adresa de revenire depusa acolo de CALL, preluand controlul la instructiunea de la aceasta adresa
- ➔ Parametrul optional n indica cati octeti o sa fie eliberiati de pe stiva
- ➔ Echivalent cu:
  - o Pop [B] ; In B se va afla astfel adresa de revenire, unde B e o eticheta din memorie (doubleword)
  - o Add ESP,[n] ; OPTIONAL
  - o Jmp [B]

COD APEL: (Cel care apeleaza functia)

- ➔ Salvare resurse volatile

- ➔ Transmitere paramentrii
- ➔ Efectuare apel cu salvare adresa de revenire (adica call sau varianta echivalenta)

COD DE INTRARE: (Adica inceputul codului din modulul apelat)

- ➔ Creare stackframe nou: (Pas necesar pentru programarea ASM+C)
  - Push EBP
  - Mov EBP,ESP
- ➔ Alocare spatiu de variabile locale SUB ESP, nr_octeti (Pas necesar pentru programarea ASM+C)
- ➔ Salvare resurse nevolatile (registrii care nu tin implicit de apel)

COD DE IESIRE: (Adica sfarsitul codului din modulul apelat)

- ➔ Restaurare registrii nevolatili
- ➔ Eliberare spatiu de variabile locale: ADD ESP, nr_octeti (Pas necesar pentru programarea ASM+C)
- ➔ Eliberare cadru stiva: (Pas necesar pentru ASM+C)
  - Mov esp, ebp
  - Pop ebp
- ➔ Revenire din subprogram (RET) si scoaterea de pe stiva a parametriilor (asta in programul principal)

OBSERVATII:

- ➔ Pentru functia: **functie(op1,op2,op3)**
  - Cod apelant:
    - Pushad; Aici salvam toti registrii (daca se folosesc in functie aka resurse volatile) (nu in cazul asta)
    - Push dword op3
    - Push dword op2
    - Push dword op1
    - Call functie
    - Add esp,4*3
    - Popad; Aici restauram registrii (daca se folosesc in functie aka restaurare resurse volatile) (nu in cazul asta)
  - Cod de intrare:
    - ; Salvare registrii folositi daca urmeaza sa fie folositi (si nu sunt explicit in parametrii functiei)
    - ; [ESP+0] este adresa de revenire
    - ; [ESP+4] este op1
    - ; [ESP+8] este op2
    - ; [ESP+12] este op3
    - …
    - ;Restaurare registrii folositi daca au fost folositi (si nu sunt explicit in parametrii functiei)
    - Ret
  - Aici nu avem nici un registru resursa VOLATILA, deci toti registrii sunt resurse NEVOLATILE, deci daca vrem sa ii folosim in subprogram trebuie salvati in SUBPROGRAM
  - Resursele VOLATILE pot fi si op1, op2, op3 daca ele sunt operanzi cu adresare la memorie, deci daca avem nevoie de valoarea lor de dinainte de apel, datoria salvarii este in programul PRINCIPAL
- ➔ Valoriile tuturor registriilor sunt transmise de la un modul la altul, deci valoarea de baza a registriilor cand au intrat in subprogram va fi valoarea lor inainte de apelul CALL, in programul principal
- ➔ Parametrii functiei pot fi transmisi:
  - Prin registrii (METODA NEINDICATA)
  - Prin stiva (METODA INDICATA) -> Cea pe care o folosim
- ➔ Daca parametrii sunt transmisi prin registrii, atunci nu mai ii punem pe stiva, si nu mai eliberam stiva la final
  - Daca parametrii unei functii sunt eax, si op1 vom avea:
    - Push op1
    - Call functie

        - Add esp,4
    - Registrii sunt transmisi automat in subprogram
    - Aici EAX este o resursa VOLATILA (deci daca o sa mai avem nevoie de valoarea lui initiala, trebuie salvat si restaurant in programul PRINCIPAL), iar restul registriilor sunt resurse NEVOLATILE, deci daca vrem sa ii folosim in subprogram, datoria salvarii si restaurarii lor, este in subprogram (EAX poate fi modificat fara a fi salvat deoarece e resursa VOLATILA, si nu e datoria subprogramului sa o salveze)