1. FLAGS

In microprocesorul 8086 Flag-urile sunt indicatoare reprezentate pe un bit. O anumita configuratie de flag-uri reprezinta un rezumat sintetic al executiei fiecarei instructiuni. Registrul EFLAGS (registrul de flag-uri) este format din 32 de biti, dar se folosesc uzual doar 9: CF (carry flag), OF (overflow flag), ZF (zero flag), SF (sign flag), PF (parity flag), DF (direction flag), TF (trap flag), IF (interrupt flag), AF (auxiliary flag).

Categoriile dupa care se clasifica flag-urile sunt urmatoarele:

i) Flag-uri ale caror valori sunt setate in functie de rezultatul ultimei operatii efectuate: CF, OF, ZF, SF, PF, AF
ii) Flag-uri ale caror valori pot fi setate de catre programator pentru a influenta modul de executie/rezultatul urmatoarei instructiuni: CF, DF, TF, IF

Cele mai des folosie flag-uri sunt CF, OF, ZF, SF.

CF (carry flag) arata daca pentru a calcula rezultatul ultimei operatii efectuate s-a realizat un transport in afara domeniului de reprezentare admis in interpretarea FARA SEMN.

Ex.:

```
mov al, 100
add al, 200
; al = 100 + 200 = 300 > 255 => CF = 1 (intervalul de reprezentare pt octeti in interpretarea fara semn este [0, 255])
```

OF (overflow flag) arata daca pentru a calcula rezultatul ultimei operatii efectuate s-a realizat un transport in afara domeniului de reprezentare admis in interpretare CU SEMN.

Ex.:

mov al, 100

add al, 200

; al = 100 + 200 = 100 + (200 – 256) = 100 – 56 = 44 > -128 si < 127 => OF = 0 (intervalul de reprezentare pt octeti in interpretarea cu semn este [-128, 127])

Pentru a obtine valoarea unui numar decimal din intervalul (127, 255] in interpretarea cu semn, il scadem din 256.

ZF (zero flag) este egal cu 1 daca rezultatul ultimei operatii efectuate este egal cu 0, 0 in caz contrar

Ex.:

mov al, 10

sub al, 10

; al = 10 – 10 = 0 => ZF = 1

SF (sign flag) este egal cu 1 daca rezultatul ultimei operatii efectuate est negativ, 0 altfel

Ex.:

mov al, 10

add al, -20

; al = 10 – 20 = - 10 => SF = 1

Instructiunile pentru accesarea directa a flag-urilor sunt:

CF: CLC (clear carry flag), STC (set carry flag), CMC (complement carry flag)

DF: CLD (clear direction flag), STD (set direction flag)

IF: CLI (clear interrupt flag), STI (set interrupt flag)

Pentru IF aceste instructiuni sunt valabile doar sub programarea pe 16 biti, astfel pe 32 de biti nu avem aceasta posibilitate

Pentru TF nu exista instructiuni de modificare deoarece acestea ar reprezenta un risc foarte mare.

Instructiuniile care iau in considerare valorile din flag-uri sunt ADC (add with carry), SBB (subtract with carry) si salturile conditionate (JZ – jump if zero flag, JNC – jump if not carry flag)

Putem modifica configuratia din registrul EFLAG prin instructiunile PUSHF si POPF, care il pun si respectiv scot de pe stiva

Flag-urile responsabile pentru situatia de depasire sunt CF si OF. Acestea sunt setate daca in urma calculului rezultatului ultimei operatii efectuate s-a realizat un transport in afara intervalului de reprezentare admis. Exista 2 astfel de flag-uri deoarece este vorba despre 2 tipuri de intrepretari: FARA SEMN (CF) si CU SEMN (OF)

2. NUMERE NEGATIVE

La nivelul microprocesorului x86 numerele negative sunt reprezentate ca si cele pozitive, doar ca bitul cel mai semnificativ (= bitul de semn) este egal cu 1. Mecanismul de reprezentare a unui numar negativ in baza 2 la nivelul microprocesorului x86 se numeste “complementul fata de 2”. Astfel, pentru a reprezenta un numar -abc in baza 2, calculam complementul fata de 2 pentru reprezentarea lui abc. Exista 4 metode de a calcula acest complement pentru un numar:

i) Scadem valoare absoluta din reprezentarea binara a lui 2^n, n reprezentand numarul de biti al reprezentarii binare pentru abc
ii) Inversam toti bitii reprezentarii binare lui abc, si adaugam 1
iii) Inversam toti bitii din stanga celui mai putin semnificativ bit cu valoarea 1, in afara de acesta

iv) Scadem valoarea absoluta a numarului din cardinalul intervalului de reprezentare admis:
Pe 8 biti: $2^8 = 256$ => [0, 255] fara semn sau [-128, 127] cu semn Pe 16 biti: $2^{16} = 65536$ => [0, 65535] fara semn sau [-32678, 32677] cu semn

Primele 3 metode ne vor da tot timpul reprezentarea in baza 2 a numarului negativ -abc, iar ultima metoda ne va da valoarea in baza 10 in functie de interpretarea aleasa a lui abc.

Ex.: -37

|-37| = 37

37(10) -> 0010 0101

37 / 2 = 18 r 1

18 / 2 = 9 r 0

9 / 2 = 4 r 1

4 / 2 = 2 r 0

2 / 2 = 1 r 0

1 / 2 = 0 r 1

-37(10) = C2(0010 0101) = 1101 1011

Ex.: -912

|-912| = 912

912(10) ->0011 1001 0000

912 / 2 = 456 r 0

456 / 2 = 228 r 0

228 / 2 = 114 r 0

114 / 2 = 57 r 0

57 / 2 = 28 r 1

28 / 2 = 14 r 0

14 / 2 = 7 r 0

7 / 2 = 3 r 1

3 / 2 = 1 r 1

1 / 2 = 0 r 1

-912(10) -> C2(0011 1001 0000) = 1100 0111 0000

Pentru a obtine valoarea in baza 10 a unei reprezentari in baza 2 in interpretarea cu semn procedam astfel:
1) Daca bitul de semn = 0, atunci valoarea in baza 10 este aceeasi, indiferent de interpretare si se calculeaza ca de obicei

2) Daca bitul de semn = 1, atunci valoarea in baza 10 este egala cu – (complementul fata de 2) in baza 10 al configuratiei data

La nivelul adunarii si scaderii, numerele negative se comporta ca si cele pozitive, singura diferenta fiind modul in care se seteaza flag-urile de depasire in urma interpretarii rezultatului fara/cu semn.

Ex.:

1) mov al, 100
   add al, 200
   ; al = 100 + 200 = 300 > 255 => CF = 1 in interpretarea fara semn
   ; al = 100 + (256 – 200) = 100 – 56 = 44 > -128 si < 127 => OF = 0 in interpretarea cu semn
2) mov al, 100
   add al, -1
   ; al = 100 + (-1) = 100 + (256 – 1) = 100 + 255 = 355 > 255 => CF = 1 in interpretarea fara semn
   ; al = 100 + (-1) = 100 – 1 = 99 > -128 si < 127 => OF = 0 in interpretarea cu semn

La nivelul inmultirii/impartirii programatorul are la dispozitie instructiuni pentru inmultirea/impartirea fara semn (mul/div) si pentru cea cu semn (imul/idiv)

Ex.:

1)
```
mov al, 32
mov ah, -255
mul ah
; ax = al * ah = 32 * (-255) = 32 * (256 – 255) = 32 * 1 = 32

mov ax, 32
mov bh, -6
div bh
; ah = ax % bh = 32 % (-6) = 32 % (256 – 6) = 32 % 250 = 32
; al = ax / bh = 32 / (-6) = 32 / (256 – 6) = 32 / 250 = 0
```

2)
```
mov al, 32
mov ah, -1
imul ah
; ax = al * ah = 32 * (-1) = -32

mov ax, 32
mov bh, -8
idiv bh
; ah = ax % bh = 32 % (-8) = 0
; al = ax / bh = 32 / (-8) = -4
```

Categoriile de instructiuni care pot opera cu numere negative in baza 2 sunt inmultirea si impartirea cu semn (imul/idiv) (prezentate mai sus) si instructiunile de conversie cu semn (cbw, cwd, cwde, cdq)

Instructiuniile de conversie cu semn extind dimensiunea din registrul corespunzator (al/ax/eax) la dimensiunea urmatoare, completand cu bitul de semn.

Ex.:

```
mov al, -1
cbw
; al = 1111 1111b = FFh
; ax = 1111 1111 1111 1111b = FFFFh
cwd
; dx:ax = 1111 1111 1111 1111 1111 1111 1111 1111b = FFFFFFFFh
cwde
; eax = 1111 1111 1111 1111 1111 1111 1111 1111b = FFFFFFFFh
cdq
; edx:eax = 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111b = FFFFFFFFFFFFFFFFh
```

3. PROGRAMARE MULTI-MODUL

Conceptele de cod de intrare, cod de apel si cod de iesire apar in contextul programarii multi-modul.

Codul de intrare are ca scop intrarea in procedura si pregatirea executiei acesteia. Sarcinile codului de intrare sunt creearea unui nou cadru de stiva (stack-frame), rezervarea pe stiva a spatiului necesar pentru alocarea variabilelor locale si salvarea pe stiva a unei copii a resurselor nevolatile modificabile. Un cadru de stiva este o structura de date stocata pe stiva care poate sa contina:

- parametrii transmisi de catre apelant
- adresa de revenire (catre instructiunea de dupa cea de apel)
- copii ale resurselor nevolatile ce pot suferi modificari
- variabile locale

Codul de apel are ca scop pregatirea si efectuarea apelului subrutinei din modulul extern. Sarcinile codului de apel sunt salvarea resurselor volatile aflate in uz, respectarea constrangerilor stabilitie si pregatirea argumentelor transmise proceduri in functie de conventia de apel stabilita (STDCALL, CDECL) si efectuarea apelului (call proc – daca va fi linkeditata STATIC, call [proc] – daca va fi linkeditata DINAMIC)

Codul de iesire are ca scop revenirea din procedura si eliberarea resurselor care nu mai sunt necesare. Sarcinile codului de iesire sunt restaurearea resurselor nevolatile care au fost modificate, eliberarea spatiului rezervat pe stiva pentru variabilele locale, dezafectarea cadrului de stiva si eliberarea parametrilor transmisi procedurii.

La nivelul limbajului de asamblare x86, aceste concepte sunt reflectate astfel:

Cod de intrare

push ebp ; salvarea resurselor nevolatile

mov ebp, esp ; crearea unui nou cadru de stiva

sub esp, 8 ; rezervarea pe stiva a spatiului necesar pentru alocarea variabilelor locale (in acest caz 8 octeti)

Cod de apel

push ecx ; salvarea resurselor volatile aflate in uz

push eax ; transmiterea parametrilor procedurii

call procedura ; efectuarea apelului + salvarea adresei de revenire

Cod de iesire

pop ebp ; restaurarea resurselor nevolatile

add esp, 8 ; eliberarea spatiului rezervat pe stiva pentru variabilele locale

mov esp, ebp ; dezafectarea cadrului de stiva

Limbajul de asamblare x86 este implicat in munca cu aceste concepte deoarece ofera posibilitatea de a avea control total asupra conventiilor de apel, procesorului si stivei, lucru esential pentru gestionarea functiilor, optimizarea performantei si interoperabilitatea intre limbaje si platforme.

In functie de limbajele de programare in care se afla subprogramele putem stabili responsabilitatile generarii codului de intrare/apel/iesire.

Astfel, daca avem apelant + apelat:

1) C + C: compilatorul C este responsabil pentru generarea tuturor 3 coduri
2) C + asm: compilatorul C genereaza codul de apel, programatorul asm genereaza codul de intrare/iesire
3) asm + C: programatorul asm este responsabil pentru generarea codului de apel, compilatorul C genereaza codul de intrare/iesire
4) asm+asm: codul de apel este reprezentate de instructiunea call (salveaza adresa de revenire si apeleaza subrutine), pentru codul de intrare nu se face nimic, iar pentru codul de iesire exista instructiunea ret

Conventia de apel CDECL este specifica limbajului C. Aceasta presupune transmiterea parametrilor subrutinei prin impingerea lor pe stiva (orice tip, dar minim DWORD, ordinea de la dreapta la stanga, numar indefinit de parametri). Resursele volatile sunt: EDX, EAX, ECX, EFLAGS. Actiunea de curatare inseamna eliberarea argumentelor de pe stiva si aceasta responsabilitate revine apelantului.

Conventia de apel STDCALL este foarte asemanatoare cu CDECL, singura diferenta fiind ca aici avem numar fix de parametri si actiunea de curatarea revine apelatului

4. OVERFLOW

Conceptul de depasire apare atunci cand rezultatul ultimei operatii efectuate depaseste intervalul de reprezentare alocat. La nivelul arhitecturii x86, microprocesorul se foloseste de flag-urile de depasire (CF si OF), fiecare avand reguli diferite de setare in functie de operatie si interpretare (FARA/CU semn).

Ex.:

<u>Adunare</u>

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

- fara semn:
  ah = 200 + 100 = 300 > 255 => CF = 1
- cu semn:

  ah = 100 + 200 = 100 + (200 – 256) = 100 – 56 = 44 => OF = 0

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

- fara semn:
  ah = 100 + 100 = 200 > 0 si < 255 => CF = 0
- cu semn:
- ah = 100 + 100 = 200 > 127 => OF = 1

<u>Scadere</u>

```
mov ah, -50
sub ah, 100
```

- fara semn:
  ah = (256 – 50) – 100 = 106 > 0 si < 255 => CF = 0
- cu semn:
  ah = -50 – 100 = -150 < -127=> OF = 1

mov ah, 100
sub ah, 101

- fara semn:
  ah = 100 – 101 = -1 < 0 => CF = 1
- cu semn:
- ah = 100 – 101 = -1 > -128 si < 127 => OF = 0

La inmultire CF = OF = 1 doar daca rezultatul nu incape pe intervalul de reprezentare al operanzilor, 0 in caz contrar (b * b = w, w * w = d, d * d = q)

Ex.:

mov al, 16

mov bl, 17

mul bl

ax = 16 * 17 = 272 > 255 deci nu incape pe byte => CF = OF = 1

mov al, 2

mov bl, 3

imul bl

ax = 2 * 3 = 6 > -128 si < 127 deci incape pe byte => CF = OF = 0

La impartire, daca rezultatul nu incape pe dimensiunea de reprezentare, programul se opreste cu run-time error (“Division by zero”), astfel valorile din flag-uri sunt irelevante.

Asamblorul ne ofera 2 instructiuni pentru a lua in considerare valoarea din flag-urile de depasire: ADC (add with carry), SBB (subtract with carry). De obicei nu se tine cont de aceasta, dar daca dorim sa adunam valori din DX:AX cu CX:BX, procedam astfel:

add AX, BX

adc DX, CX

pentru a obtine un rezultat corect.

5. ADRESE

Adresa de memorie = numarul de octeti de la inceputul memoriei RAM pana la inceputul locatiei de memorie pe care dorim sa o accesam

Ex.: in modelul de memorie flat, primul element din memorie are adresa egala cu 32 de 0 (pe 32 de biti)

Segment de memorie = diviziune logica a memoriei cu tip, limita/dimensiune si adresa de baza, succesiune de locatii de memorie menite sa deserveasca un scop similar

Ex.: code segment contine instructiuni masina (de la 1 la 15 octeti)

Offset = numarul de octeti de la inceputul segmentului curent la inceputul locatiei de memorie pe care dorim sa o accesam

Ex.: in data segment: a db 0 ; offset = 0 ; in functie de $

b db 1 ; offset = 1 ; in functie de $

Specificare de adresa = (adresa FAR) formata din segmen si offset

Ex.: mov eax [DS:a] ; in eax pune adresa far a lui a

Mecanism de segmentare = proces de impartire a memoriei in diviziuni logice, menite sa deserveasca acelasi scop

Ex.: data segment, code segment, stack segment, extra segment

Adresa liniara = baza + offset

Ex.: baza B = 1000h, offset O = 2000h => adresa liniara = B + O = 3000h

Model de memorie flat = toate adresele de baza ale segmentelor incep cu 0 (bloc continuu)

Ex.: Modul protejat x86 foloseste modelul de memorie flat

Adresa fizica efectiva = adresa obtinuta in urma procesului de segmentare in memoria fizica (hardware)

Adresa FAR = specificare completa de adresa

Moduri de specificare:

- $s_3s_2s_1s_0$ + offset
- registru_segment (CS, DS, SS, ES) + offset
- FAR [var] (unde var este de tipul qword)

Adresare directa = adresare care implica doar operanzi directi si imediati

Ex.: mov eax, dword [a]

Adresare bazata = este specificat unul dintre registrii de baza

Ex.: lea eax, [ebp]

Adresare indexata = este specificat unul din registrii de index (+ implicit scala)

Ex.: lea eax, [ebx]

Adresare indirecta = adresare care nu este directa

Ex.: mov eax, [ebx + v + 4]

Adresa NEAR = se specifica doar offsetul in segmentul curent

Ex.: mov eax, [var]

Regulile implicite de asociere intre un offset si registrul segment corespunzator sunt:

CS: call, ret, jmp | ex.: jmp acolo

DS: restul | mov eax, [ebp + ecx + 4]

SS: ESP, EBP | ex.: mov eax, [esp]

6. TIPURI DE DATE

La nivelul limbajului de asamblare, tipurile de date pot fi intelese ca dimensiunea si interpretarea datelor stocate in memorie sau in registru.

Tipurile de date comune in limbajul de asamblare sunt:

1) Tipuri de date numerice:
    - byte (8 biti)
    - word (16 biti)
    - dword (32 biti)
    - qword (64 biti)
2) Tipuri de date cu virgula mobila:
    - Stocate in FPU
3) Date de tip caracter si string:
    - Stocate ca ASCII (8 biti)
4) Adrese si pointeri

Exemple:

a db 10 ; a este de tipul byte

b dw 1; b este de tipul word

c dd 3 ; c este de tipul doubleword

d dq 15 ; d este de tipul quadword

x db ‘12345salut!’, 0 ; x este un sir de bytes terminat cu nul (sir de caractere)

y db ‘7’ ; y este un caracter, in memorie v-a fi stocat codul ascii al lui ‘7’

buffer resb 16 ; rezerva in memorie pentru buffer 16 bytes

Directivele de definire a tipurilor de date nu sunt mecanisme de definire a datelor, ci ele doar indica asamblorului cati octeti sa genereze fiecarei variabile, respectand ordinea de plasare “little-endian”.

Operatorii de tip ai operanzilor sunt necesari cand se acceseaza memoria folosind o adresa pentru a specifica asamblorului dimensiunea operandului respectiv, pentru realizarea operatiilor aritmetice cu opoeranzi din memorie si pentru conversii si mutari de date intre dimensiuni diferite.

Ex.:

move eax, dword [esi] – accesarea memoriei

add dword [ebx], 10 – realizarea unei operatii aritmetice cu operanzi din memorie

movzx eax, byte [esi] – se extinde cu zero un byte intr-un registru pe 32 de biti (conversie fara semn)

movsx eax, byte [esi] – se extinde cu semn un byte intr-un registru pe 32 de biti (conversie cu semn)

In schimb, acesti operatori de tip nu sunt necesari cand dimensiunea este clara din context, adica in cazurile: operatii intre registrii, utilizarea variabilelor definite cu dimensiune explicita si folosirea valorilor imediate.

Conversiile de tip se clasifica astfel:

i) Extindere fara semn

Ex.:

movzx eax, al ; sau succesiunea de comenzi urmatoare

mov ah, 0

mov dx, 0

push dx

push ax

pop eax

ii) Extindere cu semn

Ex.:

movsx eax, ax ; sau cwde

iii) Trunchiere

Ex.:

mov ax, eax

iv) Conversie intre signed si unsigned

Ex.:

and eax, 0FFFFFFFFh

7. ELEMENTELE DE BAZA ALE LIMBAJULUI DE ASAMBLARE

Elementele de baza ale limbajului de asamblare sunt etichetele, directivele, instructiunile si contorul de locatii.

Etichetele reprezinta nume scrise de utilizator, cu ajutorul carora se pot referi date sau zone de memorie

Instructiunile sunt scrise sub forma unor mnemonice ce sugereaza actiunea. Asamblorul genereaza octetii care codifica instructiunea respectiva

Directivele sunt indicatii date asamblorului in scopul generarii corecte a octetilor.

Contorul de locatii este un numar intreg gestionat de asamblor. Acesta reprezinta deplasamentul/offsetul in cadrul segmentului curent. (= numarul de octeti generati corect de catre asamblor la momentul curent)

Formatul unei linii sursa in limbaju de asamblare x86 este acesta:

[eticheta[:]][prefix][mnemonica][operanzi][;comentariu]

Exemple:

.aici: ; eticheta + comentariu

    rep lodsb ; prefix + mnemonica + comentariu

    add al, 10 ; mnemonica + operanzi + comentariu

a dw 12h ; eticheta + mnemonica + 1 operand + comentariu

; comentariu (ne puteam lipsi si de acesta)

Etichetele pot fi compuse doar din litere, cifre si _, $, $$, #, @, !, ., ~, si ?

Ca si prim caracter putem avea doar litere, _ , . si ?

Identificatorii NASM sunt case-sensitive, exceptie facand cuvintele cheie, mnemonicele si numele de registrii.

Exista 2 tipuri de etichete:

1) Etichete de cod – de obicei in CS, dar poate aparaea si in DS (jump-uri, apeluri de functii/subrutine)
2) Etichete de date – de obicei in DS, dar poate aparea si in CS (directive de “definire a datelor”, de fapt indica asamblorului cum sa genereze octetii)

De asemenea, exista 2 tipuri de mnemonice:

1) Directive (dirijeaza asamblorul)
2) Instructiuni (dirijeaza procesorul)

Operanzii sunt parametri care definesc valorile ce vor fi prelucrate de instructiuni sau directive. Acestia pot fi: registrii, constante, etichete, expresii, cuvinte cheie sau alte simboluri.

Operanzii pot fi specificati prin moduri de adresare. Cele trei tipuri de operanzi sunt: imediati, registru si in memorie.

Valoarea operanzilor este calculata la momentul asamblarii pt cei imediati si pt offset-urile reprezentand adresa directa, la momentul incarcarii pentru adresarea directa (adresa FAR), si la momentul executiei pentru operanzii registru si cei indirecti.

Operatorii limbajului de asamblare x86 sunt:

-, +, ~, !, *, /, //, %, %%, +, -, <<, >>, &, ^, | (in ordinea prioritatii de la cel mai prioritar la cel mai putin prioritar)

8. Reprezentarea instructiunilor masina

O instructiune masina x86 reprezinta o secventa de 1 pana la 15 octeti, care prin lor specifica o operatie de executat, operanzii asupra carora va fi aplicata, precum si modificatori suplimentari care controleaza modul in care aceasta va fi executata.

Formula:

[prefixe] + cod + [Mod R / M] + [SIB] + [deplasament] + [imediat]

Prefixele sunt constructii ale limbajului de asamblare care apar optional in componenta unei linii sursa sau a formatului intern al unei instructiuni si care modifica comportamentul standard al acelor instructiuni sau care semnaleaza procesorului modificare dimensiunii implicite de reprezentare a operanzilor si / sau a adreselor:

Exista 2 tipuri de prefixe:

1) explicite:
   - instruction prefix REP = F3h
   - segment override prefix ES = 26h
2) implicite:
   - operand override 66h
   - address override 67h

Exemple:

```
cbw ; 66:98 deoarece rez este pe 16 biti (AX)

cwd ; 66:99 deoarece rez este pe 2 registrii de 16 biti (DX:AX)

cwde ; 98 aici se respecta modul default de 32 de biti


mov edx, [bx] ; 67:8B17
```

R / M (registru / memorie):

7 6	5 4 3	2 1 0
Mod	Reg	R / M

SIB:

7 6	5 4 3	2 1 0
Scale	Index	Base

Exemple:

mov [esp + 2] , eax ; 894424 02

44h = 0100 0100 = 01 000 100

mod = 01

reg = 000

r / m = 100

24 sib

24h = 0010 0100h

scale = 00

index = 100

base = 100