ASC CURS 1 – 04.10.2023

REGISTRII = capacitati de memorare foarte mici (8, 16, 32, 64) dar foarte rapide ca viteza de acces, utilizati pt stocarea temporara a operanzilor instructiunilor cu care opereaza in mod curent un procesor.

*Procesorul lucreaza rapid doar cu nr intregi!! Calculul cu nr reale este treaba unui device separat numit coprocesor matematic.*

Octet / Byte = cea mai mica unitate de memorie adresabila

**REGISTRII GENERALI:**

EAX (registrul acumulator)

* cel mai utilizat registru din procesor
* folosit de majoritatea instructiunilor procesorului drept unul dintre operanzi in operatii aritmetice

EBX (registrul de baza) ex: a[7] a🡪baza 7🡪index

* folosit ca pointer la date

ECX (registrul contor)

* folosit in principal pt instructiuni de ciclare indicand nr de pasi ce trebuie efectuat sau pentru instructiuni care au nevoie de indicatii numerice

EDX (registru de date)

* folosit impreuna cu EAX in calculele ale caror rezultate depasesc un double word (32 biti)

ASC CURS 2 – 11.10.2023

**STIVA** = o zona de memorie in care se pot depune succesiv valori, extragerea lor facandu-se in ordinea inversa depunerii

ESP (pointer de stiva) – puncteaza spre ultimul element introdus in stiva (varful stivei)

EBP (pointer de baza) – puncteaza spre primul element introdus in stiva (baza stivei)

SS (Stack Segment)

**Exista 3 registrii de lucru cu stiva (prioritara pt procesor) si 0 registrii pt coada**

– ordinea de activare/dezactivare a functiilor dintr un program este ordinea activata de stiva, nu de coada, respectand disciplina LIFO (last in first out); spatiul de executie a oricarui program fiind stiva de executie (Run-Time Stack)

**De ce sunt 3 registrii pt lucrul cu stiva? ESP+EBP+ SS**

– baza reala a stivei ca zona de memorie este SS, iar BP este baza stack frameului principal curent

EDI (destination index) + ESI (source index) – utilizati pt accesarea elem din siruri de byte sau word

ASM nu stie ce inseamna notiunea de tablou. Sirurile trebuie sa fie simulate de catre programator prin gestionarea unei baze (inceputul sirului) si a unui index (gestionat prin EDI sau ESI).

*Orice program scris pana acum nu a fost decat o secv de instructiuni de atribuire dictata de niste instructiuni de control (if, while, repeat, for)*

FLAGURI = indicator reprezentat pe un bit

* sunt o consecinta a rezultatului ultimei operatii efectuate: **CF**, PF, AF, ZF, SF, OF (indica ce s-a intamplat)
* unele dintre ele sunt flaguri setate de catre programator anterior executiei unei instructiuni: TF, DF, IF, **CF** (indica ce urmeaza sa se intample)

**Importanta registru de flaguri**: exprima conditii / statusul unor conditii. Cand trebuie luata o decizie instructiunea curenta verifica statusul unor flaguri si in functie de valorile lor ALU (Aritmetic and Logic Unit) va decide asupra valorii conditiei respective (T/F).

EFLAGS (registru de flaguri)

* poate fi accesat prin PUSHF
* foloseste 9 biti
* rezumant sintetic al executarii fiecarei instructiuni

CF (Carry flag)

* valoarea va fi 1 in cazul in care in cadrul ultimei operatii efectuate s-a efectuat transport in afara domeniului de reprezentare a rezultatului si valoarea 0 in caz contrar
* semnaleaza depasirea in cazul interpretarii FARA SEMN

*ex:*

147 + 115 = 262 (overflow)

* + **CLC** – clear CF
  + **STC** – set CF
  + **CMC** – complement CF ( =0, dc CF = 1 / =1, dc CF = 0)

**De ce toate operatiile procesorului au max 2 operanzi?**

- pt ca cifra de transport poate avea doar valorile 0 sau 1

**De ce MUL a prevazut spatiu dublu rezultatului decat pt operanzi (asigurandu se totdeauna faptul ca rez va incapea in spatiul rezervat ), iar ADD/ SUB nu prevad un spatiu diferit pt rezultat decat dimensiunea operanzilor?**

* risipa de memorie: la sub/add ar aloca un byte/word pt un singur bit (1 cel din CF)

OF (Overflow flag)

* **De ce exista?** - pt ca sunt doua interpretari simultane (cu sau fara semn) pentru aceeasi reprezentare in baza 10 (procesorul nu stie daca interpretarea este cu sau fara semn la ADD/SUB, la MUL/DIV trebuie precizat cu IMUL/IDIV)
* semnaleaza depasirea in interpretarea CU SEMN

**Depasirea** = o conditie matematica ce exprima faptul ca

* rezultatul uoe nu a incaput in spatiul rezervat acestuia
* rezultatul nu apartine intervalului de reprezentare pe acea dimensiune de reprezentare
* operatia efectuata este un nonsens matematic in respectiva interpretare (cu semn / fara semn) si nu poate fi acceptata drept o operatie matematica corecta

In cazul impartirii, daca rezultatul obtinut nu incape in spatiul de reprezentare atunci se va semnala prin ‚Run-time error’ si unul dintre mesajele: ,Divide overflow’ ‚Division by zero’ sau ‚Zero divide’. Daca impartim un numar la zero este OVERFLOW (pt ca infinit nu incape in niciun spatiu)

PF (Parity flag)

ASC CURS 3 – 18.10.2023

* verifica daca nu s-au pierduti biti din interpretare
* valoarea PF + nr de biti 1 = un numar impar de cifre 1
* pt exemplu de mai jos PF = 1:
  + 1001 0011+
  + 0111 0011
  + = (1) 0000 0110

AF (Auxiliary Flag)

* indica valoarea transportului da la bitul 3 la bitul 4 (semi-octet)

ZF (Zero flag)

* indica daca rezultatul ultimei operatii a fost 0 (=1) sau nu (=0)

SF (Sign flag)

* indica semnul rezultatului ultimei operatii efectuate (nr negativ = 1 / nr pozitiv = 0)

TF (Trap flag)

* folosit pt scrierea/functionalitatea debugerului
* provoaca oprirea dupa o anumita instructiune prin setarea lui

Avand in vedere riscul major de setare accidentala a valorii din TF, precum si rolul sau secial in dezvoltarea de depanatoare, NU exsita instructiuni de acces direct la valoarea TF

IF (Interrupt flag)

* controlabil doar sub 16b
* ajuta la definirea sectiunilor critice in cadru programului
  + **CLI** – clear IF = 0
  + mov eax, [a]
  + .....
  + add ax, bx
  + **STI** – set IF = 1

DF (Direction flag)

* pt operare asupra sirurilor de octeti sau de cuvinte ( parcurgere inceput-sfarsit = 0 / parcurgere sfarsit-inceput = 1)
  + **CLD** – clear DF
  + **STD** – set DF

**REGISTRII DE ADRESA**

**Adresa unei locatii de memorie** *=* nr de octeti consecutivi dintre inceputul memoriei RAM si inceputul locatiei respective: *offset / deplasament*

**Specificare de****adresa** *= o* pereche formata dintr-un selector de segment si un offset

*ex:* segment : offset

**Selectorul de segment** = o valoare numerica de 16 biti care identifica (indica/selecteaza) in mod unic segmentul accesat si caracteristicile acestuia (*definit si furnizat de catre SO*)

**Offset / deplasament** = adresa unei locatii fata de inceputul unui segment / nr de octeti aflati intre inceputul segmentului si locatia in cauza

Sub 16 biti – adresa de segment era adresa fizica a inceputului respectivului segment

Sub 32 biti – adresa este predefinita/gestionata de catre SO intr-o tabela unde programatorul nu are acces (doar catre Nr. – descriptor de segment)

CS:EIP - adresa instructiunii curente de executare

CS/DS/SS/ES/FS/GS - contin selectorul de segment care identifica adresa segmentului

**Segment** = o diviziune logica a memoriei unui program, format dintr-o succesiune continua de locatii de memorie, menite sa deserveasca scopuri similare in timpul executiei

* caracterizat prin *adresa de baza* (inceput), *limita* (dimensiune) si *tip*

**Calculul de adresa** – realizat de componenta ADR din BIU

adresa de segmentare := adresa de baza + offset

ASC CURS 4 – 25 oct. 23

**COMPLEMENTUL FATA DE 2**

Matematic 🡪 reprezentarea unui numar NEGATIV in complement fata de doi este valoarea 2^n-V, unde V este valoarea absoluta a numarului reprezentat

**Var1:** (oficiala)

1 0000 0000 –

1001 0011

= 0110 1101 = 6D h = 109

**Var2:** *inversam valorile tuturor bitilor si adaugam 1*

0110 1100+

0000 0001

= 0110 1101

**Var3:** *se pastreaza toti biti de la dr la stg pana la primul bit = 1 inclusiv, iar restul se inverseaza*

1001 0011 🡪 0110 1101

**Var4:** *Regula derivata din definitia complementului fata de doi:* Suma valorilor absolute ale celor doua valori complementare in baza 10 este cardinalul multimii reprezentabile pe acea dimensiune

256 – 147 = 109

*ex:* 1001 0011 = 93h = 147 (interpretarea fara semn)

* *cat este 1001 0011 in interpretarea cu semn?* = - (complementul fata de 2) = - 109
* *147 si -109 sunt complementare dar -147 si 109 NU (unidirectionala, dar in baza 2 BIDIRECTIONALA)*

*Ex*(lucrare control) – grila cu un singur raspuns

1. Care este valoarea in **interpretarea** cu semn al lui 1001 0011? (nu a si c pt ca sunt reprezentari)

a)0110 1101 **b) -109** c) 6Dh d)147

2. valoarea cu semn al lui 93h? **b)-109**

3. valoarea cu semn al lui 147? **nu are sens pt ca 147 este deja o interpretare**

Utilizarea complementului fata de 2:

* 0xxxxx = + abc(10) 🡪 ce valoare va avea aceasta reprz in interpr cu semn = aceeasi
* 0xxxxx = +abc(10) 🡪care va fi reprz binara a valorii -abc = complem fata de 2 a lui abc
  + ex: -109 in interpr cu semn = 1001 0011 (147)
* 1xxxxx = +abc(interpr fara semn) 🡪 ce valoarea va avea aceasta reprz in interpr cu semn = -(complem fata de 2 a valorii binare initiale)
* 1xxxxx = +abc(interpr fara semn) 🡪 care va fi reprz binara a valorii -abc = cf2 a lui abc
  + Ex: -147 1001 0011 🡪 0110 1101 GRESIT
  + Reprz corecta nu poate fi reprezentata pe aceeasi dimensiune de reprezentare

**Pe n biti se pot reprezenta 2^n valori**

* Fie valorile [0,2^n – 1] in intrpr fara semn
* Fie valorile [-2^n-1, 2^n-1 – 1] in interpr cu semn
* pe 8 biti se pot reprezenta 2^8 = 256 valori [0, 255] sau [-128, 127]
* pe 16 biti se pot reprezenta 2^16 = 65.536 valori [0, 65.535] sau [-32.768, 32.767]

Intersectia [0, 255] cu [-128, 127] = [0,127] – reprezinta numerele din interpretarea fara semn + numerele pozitive din reprezentarea cu semn

1. Care este numarul min de biti pe care se poate reprz -147?

Dc n = 8, [-128, 127]

n = 9, [-256, 255] 🡪 rasp: 9 biti

1. Care e reprz in baza 16/2 pe 9 biti a lui -147?

512 – 147 = 365 = 16D h = 1 0110 1101 (fara semn)

365 = -0 1001 0011 = -093 h = -147 (cu semn)

1. Valoarea obt in reg EAX in bazele 2/10/16 in interpr fara/cu semn? (interpr cu semn se face DOAR IN BAZA 10) – 4 valori la raspuns

ASC CURS 5 – 01 nov.23

**ADRESE FAR SI NEAR**

**NEAR** = adresa in care se specifica doar offsetul, urmand ca segmentul sa fie preluat implicit dintr-un registru de segment (se afla intotdeauna in interiorul unuia dintre cele patru segmente active)

**FAR** = adresa in care programatorul indica explicit un selector de segment

* este o specificare completa de adresa si se poate exprima prin 3 moduri:
  + - specificare\_offset printr-o constanta
    - registru\_segment : specificare\_offset (registru = CS, DS, SS, ES, FS, GS)
    - FAR[variabla], unde variabila este de tip qw si contine cei 6 octeti constituind adresa FAR (variabila numita pointer)

Ex: Nume\_instructiune : [destinatie], [adresa]

Operanzi:

1. Mod registru – daca pe post de operand se afla un registru al masinii

Ex: mov **eax**, 17

1. Mod imediat – cand in instructiune se afla valoarea operandului
   1. Ex: mov eax, **17**
2. Mod adresare la memorie - cand operandul se afla undeva in memorie

In acest caz offsetul lui se calculeaza dupa urmatoarea formula:

**offset = [baza] + [index \* scala] + [constanta]**

**[baza]** : EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP

**[index**] : EAX, EBX, ECX, EDX, EBP, ESI, EDI

**scala**: fcator numeric pt a inmulti valoarea registrului index cu 1, 2, 4 sau 8

– adresare bazata : [baza]

– adresare indexata : [index \* scala]

– adresare directa : [constanta]

– adresare indirecta : [baza] + [index \* scala]

Modurile de adresare a memoriei pot fi combinate:

* Adresare direct bazata
* Adresare bazata
* Adresare scalat indexata

**ARITMETICA DE POINTERI**

a[7] = \*(a+7) = \*(7+a) = 7[a] CORECT

* Adresa oricarei variabile alocate in program este fixa pe tot parcursul executiei, iar offsetul sau este intotdeauna o constanta determinabila la momentul asamblarii/compilarii
* Adunarea/inmultirea/impartirea de pointeri nu este permisa, dar scaderea da

EX:

EBX = 17

mov eax, ebx; EAX = 17

mov eax, [ebx]; EAX = valoarea offsetului 17 = 01B9A207 (pt ca e dw)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ... | 07h | A2h | B9h | 01h | ... |
| ... | 17 | 18 | 19 | 20 | ... |

mov eax, ebx + 2; eroare de sintaxa

mov eax, [ebx + 2]

mov eax, [2\*edx – 7]

mov eax, [ecx + 4\*edx + 18]

mov eax, [ebx + edx]

mov edx, [ecx + 2\*esp + 7] ; eroare de sintaxa pt ca esp nu poate sa fie index

mov eax, [ebx\*2]; este interpretata drept [ebx + ebx ]

mov eax, [ebx\*3]; este interpretata drept [ebx + ebx \* 2]

mov eax, [ebx – edx]; eroare de sintaxa

mov eax, [23]; memory violation error

**CONTOR DE LOCATII**

a db 17, -2, 8 ..

b equ 27

c dd 123456h

lga db $-a ; corect. Se scrie imediat dupa declarare a, lga – var de memorie

lga sw $-$$; numarul de octeti total din memorie

lga dw b-a; syntax error

lga dw c-a; corect

lga dw $-a-4; corect

lg dw $-a; lenght(a) + 4

**DIRECTIVE DE DEFINIRE DE DATE**

OllyDbg – **DATA** segment incepe de la offsetul 00402000

a1 db 0,1,2,’xyx’ -> 00 01 02 ‚x’ ‚y’ ‚z’

a2 times 3 db 44h -> 44 44 44

a3 times 11 db 5,1,3 -> 05 01 03 ... (33 de octeti)

a4 dw a2 + 1, ‚bc’ -> offset(a2) + 1 = 00402009h -> 09 20 62 63

a4 dw a2 + 1, ‚b’, ‚c’ -> 09 20 62 00 63 00

a5 dd a2+1, ‚bcd’ -> 09 20 40 00 62 63 64 00

a7 db a2; SYNTAX ERROR -> obj format can only handle 16/32 relocation

a8 dw a2; 08 20

a9 dd a2; 08 20 40 00

a10 dq a2; 08 20 40 00 00 00 00 00

a11 db [a2]; EXPRESSION SYNTAX ERROR

a12 dw [a2]; EXPRESSION SYNTAX ERROR

a13 dd dword[a2]; EXPRESSION SYNTAX ERROR

a14 dq [a2]; EXPRESSION SYNTAX ERROR

a15 dd eax; EXPRESSION SYNTAX ERROR

a16 dd [eax]; EXPRESSION SYNTAX ERROR

OllyDbg – **CODE** segment incepe de la offsetul 00401000

Start:

Jmp real\_start

A db 17

B dw 1234h

C d 12345678h

Real\_start:

...

Mov eax, c; eax = 00401005

Mov edx, [c]; edx = dword ptr ds:[00401005]

...

Mov edx, [CS:c]

Mov edx, [DS:c]

Mov edx, [SS:c]

Mov edx, [ES:c]

**Efectul fiind in toate cele 5 cazuri EDX = 12345678h** deoarece toate segmentele descriu in realitate intreaga memorie, incepand de la 0 pana la capatul primilor 4gb ai memoriei

**ELEMENTE DE BAZA ALE ASM**

**Limbajul masina**

* Format din totalitatea instructiunilor masina puse la dispozitie de procesorul SC
* Reprezentate sub forma unor siruri de biti cu semnificatie prestabilita

**Limbajul de asamblare**

* Limbaj de programare in care setul de baza al instructiunilor coincide cu operatiile masinii
* Structurile sale de date coincid cu structurile primare de date ale masinii

**Elemente ale asamblorului:**

***Etichete*** = nume scrise de utilizator, cu ajutorul carora se pot referi la date sau zone de memorie

* Valoarea unei etichete este un nr intreg reprezentand adresa instruct, directivei sau datelor ce urmeaza eticheta

1. *Etichete de cod* – au scopul de a defini destinatiile de transfer ale controlului in cadrul unui program
2. *Etichete de date* – identifica simbolic unele locatii de memorie, dpdv semantic sunt echivalentul notiunii de variabila in alte limbaje

***Instructiuni*** = scrise sub forma unor mnemonice care sugereaza actiunea

* Asmablorul genereaza octetii care codifica instructiunea
* Dirijeaza procesorul

***Directive*** = indicatii date asamblorului in scopul generariii corecte a octetilor

* Dirijeaza asamblorul
* Indica modul in care sunt generate codul si datele in momentul asamblarii

Ex: relatii intre modulele obiect, definirea unor segmente, indicatii de asamblare conditionata, directive de generare a datelor

***Contor de locatii*** = numar intreg gestionat de asamblor

* Valoarea contorului coincide cu numarul de octeti generati corespunzator instructiunilor si directivelor deja intalnite in cadrul segmnetului efectiv
* Valoare apoate fi utilizata de catre programator prin simbolul **$**
* Fiecare segment are propriul sau contor de locatii

$ - indica pozitia de la inceputul expresiei (este un offset – pointer type) = ‚address of here’

$$ - indica pozitia curenta (este un offset – pointer type) = ‚address of start of current section’

$-$$ - distanta parcursa in segment ( o constanta – scalar ) = ‚current size of section’

**Operanzii** = parametrii ce definesc valorile ce vor fi prelucrate de instructiuni sau de directive

1. Imediati (val calculata in momentul asamblarii
   * Formati din date numerile constante
2. Registru (val calculata in momentul executiei)
3. In memorie

Ex: registrii, constante, etichete, expresii, cuvinte cheie sau alte simboluri

**Operatorii** – efectueaza calcule cu valori constante scalare determinabile la momentul asamblarii, cu exceptia adunarii si scaderii unei constante la un/dintr-un pointer

* 🡪 complement fata de 2 / scadere

+ 🡪 adunare

~ 🡪 prefix – complement fata de 1

! 🡪 prefix – negare logica

\* 🡪 inmultire

/ 🡪 catul impartirii fara semn

// 🡪 catul impartirii cu semn

% 🡪 restul impartirii fara semn

%% 🡪 restul impartirii cu semn

<< 🡪 deplasare biti catre stanga

>> 🡪 deplasare biti la dreapta

& 🡪 si

^ 🡪 sau exclusiv

| 🡪 sau

**Formatul unei linii sursa**

**[eticheta (:) ] [prefixe] [mnemonica] [operanzi] [;comentariu]**

*Ex:*

Aici: jmp acolo ;eticheta + mnemonica + operand + comentariu

Repz cmpsd ;prefix + mnemonica + comentariu

Start: ;eticheta+comentariu

A dw 19872, 42h. ;eticheta+mnemonica+2operanzi+comentariu

Referirea la adresa sau continut:

mov eax, et ;incarca in eax **adresa** et

mov eax, [et]. ;incarca in eax **continutul** de la adresa et

lea eax, [v]. ;incarca in eax **adresa(offsetul)** variabilei v

**Utilizarea operanzilor registru**

Mod de invocare/accesare directa: mov eax, ebx

Invocare/accesare indirecta (pt a indica locatii de memorie): mov eax, [ebx]

PUSH al (SYNTAX ERROR)

LEA (Load Effective Address) – transfera deplasamentul operandului din memorie in registru destin

Ex: lea eax, [v] = mov eax, v

Lea eax, [ebx+v-6] = mov eax, ebx+v-6 (care nu este corecta sintactic pt mov)