COD APEL / INTRARE / IEȘIRE

Apelul subrutinelor este format din trei etape: cod apel, cod de intrare și cod de ieșire. În momentul apelării unei funcții, sunt necesari anumiți pași pentru ca programul să funcționeze optim. Deși acțiunile depind în funcție de convenția de apel a subrutinei apelate, etapele rămân aceleași.

Codul de apel pregătește și efectuează apelul unei subrutine. Responsabilitățile apelantului constau în:

1. salvarea resurselor volatile (EAX, ECX, EDX, EFLAGS), presupunând că toți regiștrii își vor modifica valorile în subrutină; acest lucru se poate face cu ajutorul instrucțiunii pushad, care pune pe stivă toți regiștrii.
2. asigurarea faptului că ESP este aliniat, DF=0
3. pregătirea argumentelor pe stivă conform convenției folosind instrucțiunea push dword parametru
4. efectuarea apelului cu adresă de revenire (call)

exemplu:

```
segment code use32
start:
    mov ecx, 10
    xor eax, eax
repetă:
    push eax        } salvarea resurselor volatile
    push ecx        }
    push eax
    push dword format   } pregătirea parametrilor
    call [printf]   -> efectuarea apelului
    add esp, 2*4
    pop ecx         }
    pop eax         } restaurarea resurselor volatile
    inc eax         }
loop repetă
```

Codul de intrare este codul scris la începutul unei subrutine ap
Responsabilitatea revine apelatului de a îndeplini următoarele sarcini:

1. crearea unui cadru de stivă nouă folosind seria de instrucțiuni push ebp și mov ebp, esp

2. alocarea de spațiu pentru variabilele locale prin scăderea numărului de octeți necesari din ESP.

3. salvarea unei copii a resurselor nevolatile modificate (push)

exem

operație:

```
push ebp        } crearea cadrului de stivă
mov ebp, esp

mov eax, [ebp+8]
add eax
mov ebx, [ebp+12]
mov ecx, [ebp+16]

add eax, ebx
sub ebx, ecx

mov esp, ebp
pop ebp

ret
```

exemplu:

```
_suma_numere:
    push ebp        } crearea cadrului de stivă
    mov ebp, esp

    mov eax, [ebp+8]
    mov ebx, [ebp+12]

    add eax, ebx

    mov esp, ebp
    pop ebp

    ret
```

Codul de ieșire este codul scris la finalul unei subrutine apelate. Responsabilitatea revine apelantului de a:

1. restaura registrii nevolatili
2. elibera variabilele locale ale subrutinei
3. elibera cadrul de stivă
4. reveni din subprogram și elibera de pe stivă parametrii:

dacă este de tip CDECL, se folosește instrucțiunea ret, iar în subprogr. apelant se scrie add esp, dimensiune_argumente, iar dacă este de tip STDCALL se scrie ret dimensiune_argumente.

Exceptând resursele volatile și rezultatele directe ale funcției, starea programului după acești pași ar trebui să reflecte starea inițială

Cadrul de stivă este o structură de data stocată în stivă, de dimensiune fixă, pt. o subrutină dată, care conține parametrii pregătiți de apelant în codul de apel, adresa de revenire, copii ale surselor nevolatile folosite de acea subrutină și variabile logice.

Convenția CDECL, specifică limbajului C, implică transmiterea param. prin împingerea pe stivă în ordinea inversă declarării (de la dreapta la stânga), returnarea rezultatului în EAX sau EDX:EAX, resursele volatile EAX, ECX, EDX, EFlags și responsabilitatea apelantului de a elibera argumentele de pe stivă.

Convenția STDCALL este asemănătoare convenției CDECL, însă există două diferențe importante între cele două: convenția STDCALL impune un număr fix de parametrii, iar eliberarea argumentelor o face funcția apelată.

ASC - 31 ian 2020

CONCEPTUL DE DEPĂȘIRE

Da nivelul procesorului și al limbajului de asamblare, depășirea este o condiție matematică ce exprimă faptul că rezultatul UOE nu încape în spațiul rezervat acestuia sau rezultatul nu aparține intervalului de reprezentare admisibil pe acea dimensiune sau că operația efectuată este un nonsens matematic. Pentru a fi tratate aceste cazuri de depășire, arhitectura x86 pune la dispoziție două flaguri din registrul EFlags care semnalează depășirile:

1. Carry Flag (CF), flagul de transport, semnalează depășirea în cazul interpretării fără semn. Acesta ia valoarea 1 dacă în cadrul UOE s-a efectuat un transport în afara domeniului de reprezentare admis pe dimensiunea respectivă și 0 în caz contrar.

2. Overflow Flag (OF) semnalează depășirea în cazul interpretării cu semn. Are valoarea 1 în cazul în care în cadrul UOE s-a produs un transport, 0 în caz contrar.

I. Adunarea:

Depășirea intervine dacă una din cele două reguli de depășire la adunare se aplică: dacă suma a două nr. pozitive este negativă sau dacă suma a două nr. negative este pozitivă.

Vom analiza adunarea în interpretarea fără semn, adică vom stabili valoarea lui CF.

exemple: i)
```
mov ah, 100
mov al, 200
add ah, al
```
; CF=1, deoarece suma va fi 300, însă cum adunarea se efectuează pe 8 biți, domeniul admis de reprezentare este [0, 255], iar 300 ∉ [0, 255], deci se va efectua scăderea 300-256 = 44, obținându-se adevărata valoare a adunării (nonsens matematic)

ii) mov al, 100
mov ah, -1
add al, ah
; CF=1, deoarece -1 se transformă în nr. fără semn, 256-1=255, iar rezultatul sumei va fi 355 > 255

A doua interpretare de analizat este cea cu semn, adică vom stabili valoarea lui OF:

exemple: i) mov al, 100
mov ah, 100
add al, ah
; OF=1, deoarece 200 ∉ [-128, 127]

ii) mov al, 20
mov ah, -1
add al, ah
; OF=0, 19 ∈ [-128, 127]

II. Scăderea:

În interpretarea fără semn, CF va fi setat dacă există un împrumut de la o poziție care nu există, altfel spus dacă rezultatul nu aparține intervalului de reprezentare.

exemple: i) mov al, 100
mov ah, 101
sub al, ah
; CF=1, 100-101=-1 < 0

ii) mov al, 100
mov ah, -1
sub al, ah
; CF=1, 100-(-1) = 100-(256-1) = -155 < 0

În interpretarea cu semn, OF va fi setat în unul dintre următoarele cazuri: pozitiv - negativ = negativ sau neg - poz = poz

exemple: i) mov al, 100
mov ah, -100
sub al, ah
; OF=1; 100 - (-100) = 200 > 127

ii) mov ah, 100
mov al, 156
sub ah, al
; OF=1, 100 - 156 = 100 - (156 - 256) = 200 > 127

Necesitatea de a avea două flaguri se datorează faptului că un rezultat are două interpretări în baza 10, una cu semn și alta fără semn, fiecare având domenii de reprezentare diferite pe o dimensiune.

III. Înmulțire:

Pentru operația de înmulțire nu există o depășire reală, întrucât spațiul rezervat este dublu față de cel al operanzilor, adică byte * b = w, w * w = dw, dw * dw = qw. Sunt situații în care rezultatul înmulțirii încape în același interval cu operanzii, iar CF=OF=0. Cele două flaguri sunt setate doar dacă rezultatul încape doar pe o dimensiune dublă, cea prevăzută de sintaxa mul și imul.

exemple: i) fără semn:
mov al, -1
mov ah, 2
mul ah, al
; OF=CF, -1 * 2 = (256 - 1) * 2 = 255 * 2 > 255

ii) cu semn:
mov al, -1
mov ah, -2
imul al, ah
; OF=CF=0, întrucât -2 ∈ [-128, 127]

IV. Împărțirea:

Pt. operația de împărțire, CF și OF nu sunt definite. În cazul în care rezultatul nu încape în spațiul alocat, se produce „depășirea" la împărțire, cu efect run-time error și se emite mesajul „Division by zero"

exemple: i) mov ax, 4096
mov bl, 2
div bl
; division overflow, deoarece în al va fi câtul împărțirii, adică 2048, iar $2048 \notin [0,255]$

ii) mov ax, 20 000
mov bl, 2
idiv bl
; division overflow, $10.000 \notin [-128, 127]$

Există mai multe metode prin care programatorul poate ține cont de aceste depășiri. Asamblorul ne oferă două instrucțiuni specifice pentru adunare și scădere: ADC (add with carry) și SBB (scădere cu carry) în care se ține cont de transportul existent în flaguri. În general nu se ține cont de carry, însă atunci când avem un număr salvat în DX:AX și altul în CX:BX, adunarea celor două se face astfel:

```
add ax, bx
adc dx, cx
```

ASC - 31 ian 2020

EXPLICARE și EXEMPLIFICARE NOȚIUNI

Adresa de memorie reprezintă un identificator unic ce poziției unei locații de memorie pe care procesorul o poate accesa pt. citire sau scriere. Spre exemplu, pentru memoria flat, primul element din memorie va avea adresa 32 de 0 (pe 32 de biți).

Segmentul de memorie este o diviziune logică a memoriei unui program, caracterizată prin adresă de bază (început), limită (dimensiune) și tip. Exemple: * code segment (conține instrucțiunile mașină), data segment (date asupra cărora se acționează în conformitate cu instrucțiunile), stack segment, extra segment.

Offset sau deplasamentul reprezintă numărul de octeți adăugați la o adresă de bază / nr. de octeți aflați între începutul segmentului și locația în cauză.

Exemplu: segment data:
a db 1 -> a are offset 0 (raportat la $$)
b dw 2 -> b are offset 1

Specificare de adresă este o pereche formată dintre un selector de segment și un offset. Exemplu: mov eax, [DS:a] -> pune adresa EAX a lui a

Mecanismul de segmentare este un proces de împărțire a memoriei în diviziuni fizice, menite să deservească același scop. Exemple: * cel de la segmentul de memorie.

Adresa liniară este formată din bază + offset.

$a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 : b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 + 0_7 0_6 0_5 0_4 0_3 0_2 0_1 0_0$

exemplu: avem baza 2000h și offset 1000h = adresa liniară 3000h

Model de memorie flat = fi toate segmentele încep de la 0 (bloc continuu). Exemplu: modul protejat x86 utilizează modelul de memorie flat.

Adresa fizică efectivă reprezintă rezultatul final al segmentării în memoria fizică

Adresa FAR - se indică explicit un selector de segment, așadar adresa FAR este o specificare completă de adresă care poate fi specificată concret printr-o variabilă constantă, printr-un registru-segment : specificare-offset sau FAR [variabilă] (conține 6 octeți care construiesc adresa)

Adresa NEAR este formată doar din offset, segmentul se adaugă implicit. Exemplu: mov eax, [v]

Adresă bazată = intervin regiștrii de bază

Exemplu: mov eax, [ebx]

Adresă directă = implică doar operanzi direcți și imediați

exemplu: mov eax, [a+4]

Adresare indirectă = care nu o directă

exemplu: mov ax, [ebx+v+4]

Adresare indexată = intervin regiștrii index (și implicit scală)

exemplu: mov eax, [2*eax]

Reguli implicite de asociere între offset și reg. segment:

- CS: pt. jmp, call, ret ex: jmp eticheta
- SS: în adresări sib ce folosește EBP sau ESP drept bază
  ex: mov ax, [ESP]
- DS: pt. restul accesărilor de date
  ex: mov ax, [EBP+ECX+4]

ASC - 1P feb 2019

FLAGURI :

I. Un flag este un indicator reprezentat pe un bit, iar configurația registrului de flaguri oferă un rezumat sintetic al execuției fiecărei instrucțiuni. Arhitectura x86 dispune de acest registru, registrul EFLAGS, reprezentat pe 32 de biți, din care doar nouă sunt utilizați în mod uzual: CF, OF, ZF, PF, IF, DF, SF, AF, TF. Acestea se împart în două categorii: flaguri setate ca urmare a ultimei operații efectuate (CF, OF, ZF, SF, AF, PF) și flaguri setate ulterior de programator (CF, DF, TF, IF)

1. Carry Flag (CF) este un flag de transport și semnalează depășirea în cazul interpretării fără semn în baza 10. Acesta ia valoarea 1 dacă în cadrul UOE s-a efectuat un transport în afara domeniului de reprezentare admis pe dimensiunea respectivă și 0 în caz contrar. Ulterior, CF este un flag care poate fi setat de programator, întrucât dispunem de două instrucțiuni pentru calculul cu transport: ADC (add with carry) și SBB (scădere cu carry). Limbajul de asamblare pune la dispoziție 3 instrucțiuni de setare al acestui flag: CLC (clear CF, CF=0), STC (set CF, CF=1) și CMC (complement CF, CF=1 → CF=0 sau CF=0 → CF=1).

exemple: i)
```
mov ah, 100 ; ah = 100
mov al, 200 ; al = 200
add ah, al
; CF = 1, întrucât 100+200=300>255,
  deci nu aparține intervalului admis pe octet
```

ii)
```
mov al, 100 ; ...
mov ah, -1  ; ...
add al, ah
; CF = 1, întrucât 100+(-1) = 100+(256-1) =
  = 356 > 255
```

Ca și regulă generală, un număr negativ se transformă în număr fără semn prin scăderea acelui număr din cardinalul intervalului de reprezentare; în cazul unui octet, intervalul de reprezentare este $[0, 255]$, iar cardinalul este 256.

exemple: i) mov al, 100
mov ah, 101
sub al, ah
; CF = 1, întrucât $100 - 101 = -1 < 0$

ii) mov al, 100
mov ah, -1
sub al, ah
; CF = 1, întrucât $100 - (-1) = 100 - (256 - 1) =$
$= -155 < 0$

2. Overflow Flag (OF) este tot un flag de transport, însă acesta semnalează depășirea în cazul interpretării cu semn. Are valoarea 1 în cazul în care în cadrul UOE s-a produs un transport în afara domeniului de reprezentare al rezultatului și 0 în caz contrar.

Ca și regulă generală, pe octet, dacă un număr aparține domeniului $(127, 255]$, acesta se aduce în intervalul de reprezentare admis de un octet prin scăderea cu 256. Această operație nu schimbă CF / OF pe 00

exemple (add): i) mov al, 100
mov ah, 100
add al, ah
; OF = 1, pentru că $100 + 100 = 200 > 127$

ii) mov al, -100
mov ah, 156
add al, ah
; OF = 1, întrucât $-100 + 156 = -100 + (156 - 256) =$
$= -100 - 100 = -200 < -128$

exemple (sub): i) mov al, 100
mov ah, -100
sub al, ah
; OF = 1, 100 + 100 = 200 > 127

ii) mov ah, 100
mov al, 156
sub ah, al
; OF = 1, 100 - 156 = 100 - (156 - 256) = 100 + 100 = 200 > 127

Necesitatea de a avea două flaguri a se ocupă cu determinarea cazurilor de depășire se datorează faptului că un rezultat are două interpretări în baza 10, una cu semn și alta fără semn, fiecare având domenii de reprezentare diferite pe o dimensiune.

Pentru operația de înmulțire, nu există o depășire reală, întrucât spațiul rezervat este dublu față de cel operanzilor, adică byte * byte = word, word * word = dword, dword * dword = qw

exemple: i) fără semn:
mov al, -1
mov ah, 2
mul al, ah
; OF = CF = 1, întrucât -1 * 2 = (256 - 1) * 2 = 255 * 2 > 255

ii) cu semn:
mov al, -1
mov ah, 2
imul al, ah
; OF = CF = 0, întrucât -2 ∈ [-127, 128]

Pentru operația de împărțire, CF și OF nu sunt definite. În cazul în care rezultatul nu încape în spațiul alocat, se produce „depășirea" la împărțire, cu efect run-time error și se emite o eroare de tipul „Division by zero".

3. Parity Flag (PF) este folosit, de cele mai multe ori, în transmisii de date pentru a nu se pierde biții. Acesta verifică dacă există un număr impar de biți 1 în octetul cel mai puțin semnificativ al reprezentării rezultatului UOE.

4. Auxiliary Flag (AF) reține valoarea transportului de la bitul 3 la bitul 4

5. Zero Flag (ZF) indică dacă rezultatul UOE este 0 sau NU. Acest flag nu se setează la înmulțire și împărțire indiferent de rezultat. Astfel, pentru adunare și scădere, ZF = 1 dacă rezultatu UOE este 0 și ZF=0 în caz contrar.

exemple: i) mov al, 15
mov ah, 241
add al, ah
; ZF=1, deoarece 241+15=256=256-256=0
(256 nu încape pe un byte, deci se salvează doar ultimii 8 biți)

6. Sign Flag (SF) indică bitul de semn al rezultatului UOE. Rezultatul va fi adus la interalul de reprezentare admisibil. Astfel, SF=1 dacă rezultatul UOE este strict negativ și SF=0 dacă este pozitiv.

exemple: i) mov al, 10
mov ah, -1
add al, ah
; SF = 0, pt. că 10-1 = 9 ∈ [-128, 127], 9 > 0

ii) mov al, -2
mov ah, -3
add ah, al
; SF = 1, -2-3 = -5 < 0

7. Trap Flag (TF) este un flag de depanare, folosit la scrierea unui debugger. Acesta nu poate fi modificat de către programator pentru evitarea setării accidentale a valorii lui. Dacă TF = 1, atunci mașina se oprește după fiecare instrucțiune

8. Interrupt Flag (IF) este un flag de întrerupere folosit pt. secțiuni critice: când IF = 1, se oprește rularea oricărui alt proces. Deși nu poate fi setat sub 32 de biți, acesta putea fi modificat de către programator la programarea sub 16 biți cu ajutorul a două instrucțiuni: CLI (clear IF, IF = 0) și STI (set IF, IF = 1).

9. Direction Flag (DF) este un flag de direcție utilizat, de cele mai multe ori, la lucrul cu șiruri. Dacă DF = 1, atunci șirul s-ar parcurge de la dreapta la stânga, altfel șirul s-ar parcurge de la stânga la dreapta. Programatorul poate modifica valoriile acestor două flaguri cu ajutorul instrucțiunilor CLD (clear DF, DF = 0) și STD (set DF, DF = 1).

Limbajul de asamblare pune la dispoziție două instrucțiuni de transfer de uz general pentru registrul EFlags și anume PUSHF / PUSHFD pentru depunerea registrului pe stivă și POPF / POPFD pentru extragerea vârfului stivei și stocarea în EFlags.

Folosind cele două instrucțiuni, putem memora registrul ett EFlags în EAX, urmând ca programatorul să poată manipula valoriile din flagurile după cum dorește:

```
PUSHF   ; se pune EFlags pe stivă
POP EAX ; se extrage vf. stivei și se pune în EAX
...     ; instrucțiuni de modificare a valorilor
PUSH EAX
POPF
```