Complementul față de 2 – analiză ... (a picat la examen... mai mare ghinionu n aveam)

**Conceptul de depășire**

**CF *(Carry Flag)*** este flagul de transport. Are valoarea 1 în cazul în care în cadrul ultimei operații efectuate (UOE) s-a efectuat transport în afara domeniului de reprezentare a rezultatului și valoarea 0 în caz contrar. Flagul CF semnalează depășirea în cazul interpretării FĂRĂ SEMN.

**OF *(Overflow Flag)*** este flagul pentru depășirea CU SEMN. Dacă rezultatul UOE în interpretarea CU SEMN a operanzilor nu a început în spațiul rezervat operanzilor (intervalul de reprezentare admisibil), atunci acest flag va avea valoarea 1, altfel va avea valoarea 0.

**Regiștrii de adresă și calcul de adresă**

**Adresa unei locații** este numărul de octeți consecutivi de la începutul memoriei RAM până la locația respectivă.

**Segmentul** este o succesiune continuă de locații de memorie, menite să deservească scopuri similare în timpul execuției unui program.

**Offset** este adresa unei locații față de începutul unui segment sau numărul de octeți dintre începutul segmentului și locația în cauză. Un offset se consideră valid dacă nu depășește limita segmentului la care se raportează.

**Adresa logică** este o pereche formată dintr-un selector de segment și un offset.

**Selectorul de segment** este o valoare numerică de 16 biți care identifică în mod unic segmentul accesat și caracteristicile acestuia. Un selector de segment este definit și furnizat de către sistemul de operare.

O **adresă liniară** ***(adresă de segmentare)*** este rezultatul calculului:

**a7a6a5a4a3a2a1a0 := b7b6b5b4b3b2b1b0 + o7o6o5o4o3o2o1o0**

O adresă logică se numeste și **adresă FAR**. Dacă o adresă se precizează doar prin offset o numim **adresă NEAR**.

Calculul adresei liniare este efectuat de **ADR** și **BIU**.

Acest mod de utilizare a segmentării se mai numește **model de memorie flat**.

Arhitectura x86 conține 4 tipuri de segmente diferite:  
- **CS *(code)*** : segment de cod (conține instrucțiuni mașină)  
- **DS *(data)*** : segment de date (conține date asupra cărora se acționează)  
- **SS *(stack)*** : segment de stivă  
- **ES *(extra)*** : segment suplimentar de date

**Regiștrii CS, DS, SS, ES** din **BIU** conțin valorile selectorilor de segment active.

**Regiștrii FS și GS** pot conține selectori care indică segmente suplimentare.

**Registrul EIP** conține offsetul instrucțiunii curente din **segment de cod**.

**Reprezentarea instrucțiunilor mașină**

O instrucțiune x86 reprezintă o operație specifică de executat, are maxim 2 operanzi care în majoritatea operațiilor poartă numele de **sursă** și **destinație**. Maxim unul poate fi din memoria RAM, celălalt aflându-se într-un registru.

**nume\_instrucțiune destinație, sursă**

Formatul unei instrucțiuni este variabil și poate ocupa între 1 și 15 octeți.

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

**Prefixele** controlează modul în care o instrucțiune se execută.

Octetul **Mod R/M** ***(mod registru / memorie)*** specifică pentru unele instrucțiuni natura și locul operanzilor (registru sau memorie).

**offset = [bază] + [index \* scală] + [constantă]**

**bază** – valoarea unui registru (EAX, EBX, ECX, EBP, ESI, EDI)  
 **index** – valoarea unui registru (EAX, EBX, ECX, EBP, ESP, ESI, EDI)  
 **scală** – 1, 2, 4, 8

**Deplasamentul** apare în cazul unor forme de adresare particulare.

**Valoarea imediată** oferă posibilitatea definirii unui operand ca fiind o constantă numerică pe 1, 2 sau 4 octeți.

**Adrese FAR și NEAR**

Pentru o locație din memoria RAM sunt necesare 2 valori: una care să indice segmentul, alta care să indice offsetul din cadrul segmentului.

**Adresa NEAR** specifică doar offsetul, segmentul fiind luat implicit dintr-un registru de segment.

**Adresa FAR** conține un selector de segment indicat explicit de programator.

**Arhitectura microprocesorului 80x86**

Microprocesorul este format din 2 componente principale:  
- **EU** ***(executive unit)*** : execută instrucțiuni mașină prin intermediul componentei **ALU** ***(Arithmetic and Logic Unit)***  
- **BIU** ***(bus interface unit)*** : pregătește instrucțiunile mașină să fie executate

**EU** și **BIU** lucrează în paralel – în timp ce EU execută instrucțiunea curentă, BIU o pregătește pe următoarea.

**Regiștri generali EU**

- **EAX** – **registrul acumulator**, folosit de majoritatea instrucțiuniilor ca fiind  
 unul dintre operatori  
- **EBX** – **registru de bază** (base register, cu acest rol pe 16 biți)  
- **ECX** – **registru contor** pentru instrucțiunile care au nevoie de indicații  
 numerice  
- **EDX** – **registru de date**, împreună cu EAX se folosește în calcule ale căror  
 rezultat depășesc un dublucuvânt  
- **ESP, EBP** – **regiștrii destinați lucrului cu stiva**, stiva fiind o zonă de memorie  
 în care se pot depune și extrage valori  
- **ESP** – **(Stack Pointer)** – punctează spre elementul din vârful stivei (ultimul  
 element introdus în stivă)  
- **EBP** – **(Base Pointer)** – punctează spre elementul de la baza stivei (primul  
 element introdus în stivă)  
- EDI, ESI – **regiștrii de index** utilizați pentru a accesa elemente din șiruri de  
 octeți sau cuvinte **(Source Index, Destination Index)**

**Flagurile**

Un flag este un indicator reprezentat pe un bit. O configurație a registrului de flaguri indică un rezumat sintetic a execuției fiecărei instrucțiuni. Pentru x86, registrul EFLAGS are 32 de biți din care sunt folosiți uzual doar 9.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X 31 | X 30 | ... | OF 11 | DF 12 | IF 9 | TF 8 | SF 7 | ZF 6 | X 5 | AF 4 | X 3 | PF 2 | X 1 | CF 0 |

**CF** ***(Carry Flag)*** este flagul pentru transport. Are valoarea 1 în cazul în care în cadrul UOE s-a efectuat transport în afara domeniului de reprezentare și 0 în caz contrar. Flagul CF semnalează depășirea în interpretarea FĂRĂ SEMN.  
 **PF *(Parity Flag)***– valoarea lui se stabilește împreună cu numărul de biți de 1 din octetul cel mai puțin semnificativ al reprezentării UOE să rezulte un număr impar de cifre de 1.  
 **AF *(Auxiliary Flag)*** – indică valoarea transportului de la bitul 3 la bitul 4 al UOE.  
 **ZF *(Zero Flag)*** – primește valooarea 1 dacă rezultatul UOE este egal cu 0 și valoarea 0 dacă rezultatul UOE este diferit de 0.  
 **SF *(Sign Flag)*** – primește valoarea 1 dacă rezultatul UOE are bitul de semn egal cu 1 și valoarea 0 în caz contrar  
 **TF *(Trap Flag)*** este un flag de depanare. Dacă are valoarea 1, atunci mașina se oprește după fiecare instrucțiune.  
 **IF *(Interrupt Flag)*** este flag de întrerupere.  
 **DF *(Direction Flag)*** pentru operarea șirurilor de octeți sau de cuvinte. Dacă are valoarea 0, deplasarea se face de la început spre sfârșit. Iar pentru valoarea 1 deplasarea se face de la sfârșit spre început.  
 **OF *(Overflow Flag)*** este flag pentru depășire CU SEMN. Dacă rezultatul UOE în interpretarea CU SEMN a operanzilor nu a încăput în spațiul rezervat operanzilor, atunci flagul va avea valoarea 1, 0 în caz contrar.

**Instrucțiuni pentru setarea flagurilor**:  
CLC 🡪 CF = 0 CLI 🡪 IF = 0  
STC 🡪 CF = 1 STI 🡪 IF = 1 (acestea 2 doar pe 16 biți)  
CMC 🡪 complementarea CF  
CLD 🡪 DF = 0  
STD 🡪 DF = 1

**Elemente de bază ale limbajului de asamblare**

**Limbajul mașină** al unui sistem de calcul (SC) este format din totalitatea instrucțiunilor mașină puse la dispoziție de procesorul SC. Acestea se reprezintă sub forma unor șiruri de biți cu semnificație prestabilită.

**Limbajul de asamblare** al unui calculator este un limbaj de programare în care setul de bază al instrucțiunilor coincid cu operațiile mașinii și ale cărui structuri de date coincid cu structurile primare de date ale mașinii.

Elementele cu care lucrează un asamblor sunt:  
- **etichete** = nume scrise de utilizator, cu ajutorul cărora se pot referi date sau  
 zone din memorie  
- **instrucțiuni** = scrise sub forma unor mnemonice care sugerează acțiunea.  
 Asamblorul generează octeții care codifică instrucțiunea  
 respectivă  
- **directive** = sunt indicații date asamblorului în scopul generării corecte a  
 octeților  
- **contor de locații** = este un număr întreg gestionat de asamblor, care în fiecare  
 moment conține numărul de octeți generați (simbol ‘$’)

**Formatul unei linii sursă în limbaj de asamblare x86**

**[etichetă[:]] [prefixe] [mnemonică] [operanzi]**

ex: aici: jmp acolo ; etichetă + mnemonică + operand + comentariu  
 repz cmpsd ; prefix + mnemonică + comentariu  
 start: ; etichetă + comentariu  
 a div 197, 42h ; etichetă + mnemonică + 2 operanzi + comentariu

**p – adresa variabilei p  
 [p] – valoarea variabilei p**

**Expresii**:  
 Sunt evaluate în momentul asamblării pentru operanzii imediați și pentru cei cu adresare directă.

**Operanzii**:  
 Se împart în 3 categorii:  
⬩ imediați  
⬩ registru  
⬩ din memorie

**Operanzii imediați** sunt formați din date numerice constante calculabile la momentul asamblării. Aceștia pot fi reprezentați binar (a db 1001 0101b), octal (a db 0o1246), decimal (a db 123) și hexazecimal (a db 0A2h).

**Operanzii registru** pot fi accesați direct și indirect:  
- direct: mov EAX, EBX  
- indirect: mov EAX, [EBX]

**Operanzii din memorie** se împart în 2 categorii: cu adresare directă și cu adresare indirectă.

Cei cu **adresare directă** sunt fie o constantă, fie un symbol care reprezintă adresa unor instrucțiuni sau a unor date. Deplasamentul operanzilor cu adresare direct se calculează în momentul asamblării.

Operanzii cu **adresare indirectă** utilizează regiștrii pentru a indica adresarea din memorie. Deplasamentul se calculează după formula:  
 deplasament = [reg\_de\_bază] + [reg\_index \* scală] + [constantă]

**Operatorii** sunt utilizați pentru combinarea, modificarea și analiza operanzilor. Operatorii efectuează calculi cu valori constante determinabile la momentul asamblării.

Instrucțiunile efectuează calculele cu valor ice pot fi necunoscute până în momentul execuției.

Operatorii se impart în:  
- **operatori de deplasare pe biți** (‘**<<**’, ‘**>>**’)  
ex: mov AH, 01110111b << 3 ; 1011100b  
- **operatori logici pe biți**: ~ (completare biți); & (și bit cu bit); | (sau bit cu bit);  
 ^ (xor bit cu bit (sau exclusiv))  
- **operatori de specificare a segmentului**: aceștia comandă calcularea adresei FAR a unei variabile sau etichete în funcție de un anumit segment

**sintaxa**: segment: expresie  
 **ex**: [ss : ebx + 4] [es : 082h]  
- **operatori de tip**: specifică tipurile unor expresii și a unor operanzi păstrați în memorie  
 **sintaxa**: tip expresie  
 unde specificatorul de tip este unul dintre cuvintele cheie: **BYTE**, **WORD**, **DWORD**, **QWORD**

**Directive**

**Directivele** indică modul în care sunt generate codul și datele în momentul asamblării.

**Directiva segment** permite direcționarea octeților de cod sau date înspre segmental precizat, care poartă un nume și are associate diverse caracteristici.

**SEGMENT nume [tip] [ALIGN=aliniere] [combinare] [utilizare] [CLASS=clasă]**

**Directive pentru definirea datelor  
definire date** = **declarare** + **alocare**  
(proces unic) (specificarea atributelor, (rezervarea spațiului de memorie  
 poate apărea de mai multe ori) necesar, proces unic)

**Directiva TIMES** permite asamblarea repetată a unei instrucțiuni sau definiții de data (ex: Tabchar TIMES 80 DB ‘a’, TIMES 32 add EAX, EDX)

**Directiva EQU** permite atribuirea, în faza de asamblare, unei valori numerice sau șir de caractere unei etichete fără alocare de spațiu de memorie sau generare de octeți.

**Instrucțiuni ale limbajului de asamblare**

**Salturi**  
 Se impart în:  
- **NEAR**  
- **FAR**  
- **PRIN ETICHETE**

**Salturile NEAR** se pot realiza prin oricare tipuri de operanzi (etichetă, registru, cu adresare la memorie)

**Salturile FAR** (se modifică atat valoarea din EIP, cât și cea din CS) se pot realiza doar prin intermediul unui operand de adresare la memorie pe 48 de biți (pointer FAR)

**Prin etichete**, chiar dacă se sare într-un alt segment nu se va considera că este un salt FAR deoarece nu se modifică CS-ul (din cazua FLAT MEMORY MODEL). Se va modifica doar EIP-ul și saltul se consider ca fiind un salt NEAR.

**Prin regiștri nu este posibil FAR** deoarece regiștrii sunt pe 32 de biți și se poate astfel specifica drept operand al unui JMP doar un offset (salt NEAR), deci e imposibil de a preciza un salt FAR cu un operand limitat la 32 de biți.

Ca și **instrucțiuni de salt condiționat** avem:  
- CF = 1: **JB** (Jump Bellow), **JNAE** (Jump Not Above or Equal), **JC** (Jump Carry)  
- CF = 0: **JAE** (Jump Above or Equal), **JNB** (Jump Not Bellow), **JNC** (Jump Not Carry)  
- CF = 1 sau ZF = 1: **JBE** (Jump Bellow Equal), **JNA** (Jump Not Above)  
- CF = 0 și ZF = 0: **JA** (Jump Above), **JNBE** (Jump Not Bellow Equal)  
- ZF = 1: **JE** (Jump Equal), **JZ** (Jump Zero)  
- ZF = 0: **JNE**, **JNZ**  
- SF ≠ OF: **JL**, **JNGE**  
- SF = OF: **JGE**, **JNL**  
- ZF = 1 sau SF ≠ OF: **JLE**, **JNG**  
- ZF = 0 și SF = OF: **JG**, **JNLE**  
- PF = 1: **JP**, **JPE**  
- PF = 0: **JNP**, **JPO**  
- SF = 1: **JS**  
- SF = 0: **JNS**  
- OF = 1: **JO**  
- OF = 0: **JNO**

**Prefixe**  
 Arhitectura x86 poate avea până la 4 prefixe.  
Prefixe precizate explicit de către programator:  
- F3h = **REP** (repetă bucla de ECX ori), **REPE** (bucla se termină dacă ZF = 0)  
- F2h = **REPNE** (bucla se termină daca ZF = 1)

**Instrucțiuni de transfer: MOV, PUSH, POP, XCHG, XLAT, PUSHAD, POPAD, PUSHFD, POPFD.**

Dacă operandul destinație al instrucțiunii **MOV** este unul dintre cei 6 regiștrii de segment, atunci sursa trebuie să fie unul dintre cei 8 regiștrii generali ai UE.

Instrucțiunile **PUSH** și **POP** repun, respective extrag operanzi de tip dublucuvânt din stivă.

**XCHG** permite interschimbarea conținutului a 2 operanzi de aceeasi dimensiune (**XCHG op1, op2**).

**XLAT** “traduce” octetul din AL într-un alt octet, utilizând astfel o tabelă de corespondență (**[reg segment] XLAT**). Efectul lui XLAT este de a înlocui octetul din AL cu octetul din tabelă (din EBX) cu numărul de ordine de la valoarea din AL.

**PUSHAD** și **POPAD** depun pe stivă și extrag de pe stivă toți regiștrii (EAX, ECX, EDX, EBX, EBP, ESP, ESI, EDI).

**PUSHFD** depune EFLAGS pe stivă.  
 **POPFD** extrage vârful stivei și îl depune în EFLAGS.

Limbajul de asamblare pune la dispoziție niste instrucțiuni de setare a flagurilor: CLC (CF = 0)  
 CMC (CF = ~CF)  
 STC (CF = 1)  
 CLD (DF = 0)  
 STD (DF = 1)

**Instrucțiuni de conversie**

**CBW** – conversie octet conținut în **AL** la cuvântul **AX**  
**CWD** – conversie cuvânt conținut în **AX** la dublucuvânt în **DX:AX**  
**CWDE** – conversie cuvânt conținut în **AX** la dublucuvânt în **EAX**  
**CDQ** – conversie dublucuvânt conținut în **EAX** la quadword în **EDX:EAX**  
 Aceste sunt utilizate în conversia cu semn (completează partea superioră cu bitul de semn), pe cand pentru conversia fără semn se realizează zerorizarea octetului sau cuvântului superior.

**MOVZX d, s** – încarcă în d (registru) de dimensiune mai mare decât s (registru  
 sau operand din memorie) conținutul lui s FĂRĂ SEMN  
**MOVSX d, s** – încarcă în d (registru) de dimensiune mai mare decat s (registru  
 sau operand din memorie) conținutul lui s CU SEMN

**Instrucțiuni aritmetice**

- **operații logice pe biți** (AND, OR, XOR și NOT)  
- **deplasări și rotiri pe biți** (SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR)  
- **comparații între operanzi** (CMP d, s; TEST d, s)  
 CMP d, s – execuție fictivă d - s OF, SF, ZF, AF, PF, CF  
 TEST d, s – execuție fictivă d AND s OF = 0; CF = 0  
 SF, ZF, PF modificați  
 AF – nedefinit  
 **Instrucțiuni de ciclare** (LOOP, LOOPE, LOOPNE, JECXZ)  
 LOOP – se decrementează ECX, după se compară cu 0 și salt dacă ECX ≠ 0  
 LOOPE – se decrementează ECX, se oprește dacă ECX = 0 sau ZF = 0  
 LOOPNE – se decrementează ECX, se oprește dacă ECX = 0 sau ZF = 1  
 JECXZ – realizează saltul dacă ECX = 0  
 **Instrucțiunile CALL și RET**  
 CALL – apelarea unei procedure  
 RET [n] – semnifică terminarea execuției secvenței apelate și revenire  
 – eliberează n octeți din stivă aflați sub adresa de revenire

**Programare multimodul**

**Cod apel, intrare, ieșire**

**Cod apel (apelatorul)**:  
- salvare resurse volatile (EAX, ECX, EDX, EFLAGS)  
- transmitere parametrii  
- efectuare apel cu salvarea adresei de revenire

**Cod de intrare (apelatul)**:  
- creare stackframe sau (**PUSH** EBP **MOV** EBP, ESP)  
- alocare spațiu variabile locale: SUB ESP, nr\_octeți  
- salvare resurse nevolatile posibil a fi modificate

**Cod de ieșire (apelatul)**:  
- restaurare regiștrii nevolatili  
- eliberare spațiu variabile locale (ADD ESP, nr\_octeți\_var\_locale)  
- eliberare cadru de stivă (**MOV** ESP, EBP **POP** EBP)  
- revenirea din subprogram (**RET**) și scoaterea de pe stivă a parametrilor

**Responsabilități generare cod apel-cod intrare-cod ieșire**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Apelant | Apelat | Cod apel | Cod intrare | Cod ieșire |
| C | C | Compilator C | Compilator C | Compilator C |
| C | ASM | Compilator C | Programator ASM | Programator ASM |
| ASM | C | Programator ASM | Compilator C | Compilator C |
| ASM | ASM | CALL reluarea adresei de revenire | NIMIC ramane la latitudinea programatorului | RET recuperarea adresei de revenire + salt |

**Resurve volatile și nevolatile**

**Resursele volatile** sunt reprezentate de către acei regiștrii pe care convenția de apel îi definește ca aparținând subrutinei apelate, ca atare apelantul având datoria, ca parte a codului de apel, de a salva valoarea acestora (dacă-I folosește), iar apoi să le restaureze la valorile vechi la ieșirea din apel. Deci, apelantul salvează resursele volatile (ca parte a codului de apel) și tot el le restaurează după apel în cadrul codului current, dar nu ca parte a vreunui cod de apel / intrare / ieșire.

**Resursele nevolatile** sunt orice adrese de memorie ori regiștrii care nu aparțin explicit subrutinei chemate, iar dacă acesta dorește să efectueze modificări, este necesar să le salveze la intrare (ca parte a codului de intrare), iar apoi să le restaureze inainte de ieșire la valorile lor originale (ca parte a codului de ieșire).

**Diferența dintre cele 2 tipuri de resurse** este aceea că **la cele volatile apleantul are datoria să aiba grijă de valorile regiștrilor în cauză**, pe când **la cele nevolatile apelantul nu are nicio datorie**, fiindu-I garantat că valorile se păstrează, codul apelat fiind cel responsabil să restaureze la sfârșit orice modificare.

**STDCALL**