Capitolul 5. Tehnici de programare în limbaj de asamblare.

Acest capitol este dedicat tehnicilor de programare în limbaj de asamblare., adică modalităților de proiectare și implementare a modulelor de program. Deoarece codul sursă scris în asamblare este greu de urmărit este esențială o abordare sistematică și ordonată a dezvoltării programelor. Utilizarea disciplinată a procedurilor, a mecanismelor standard de transfer de parametri și a macroinstrucțiunilor contribuie esențial la obținerea de programe clare, eficiente și ușor de întreținut.

În esență, specificarea modulelor de program în asamblare nu diferă de cea specifică limbajelor de nivel înalt, în sensul că se pornește de la o descriere abstractă a algoritmului care trebuie implementat. O problemă specifică este asignarea variabilelor. Dacă într-un limbaj de nivel înalt acest lucru nu creează probleme (introducem câte variabile dorim, fără a ne pune problema spațiului alocat), în asamblare trebuie să asignăm explicit variabilele. Prima modalitate este să asignăm cât mai multe variabile în regiștrii procesorului. Deoarece numărul acestora este limitat, vom fi nevoiți să asignăm variabile și în segmentul de date (statice) sau pe stivă. Pentru a nu crește numărul variabilelor peste o limită rezonabilă, este esențial ca modulele de program să implementeze subprobleme de dimensiuni adecvate (nu foarte mari), ceea ce implică o descompunere a problemei inițiale în subprobleme bine specificate [2].

Majoritatea operațiilor de bază din programarea structurată (decizia, selecția, ciclurile cu test anterior și cele cu test posterior) implică în mod inerent evaluarea unor condiții logice. În asamblare, aceste condiții sunt în general de tip comparație între valori numerice.

5.1. Structuri de control

De obicei, un program nu se limitează la o secvență liniară de instrucțiuni. În timpul procesului său se poate bifurca, repeta sau ia decizii. În acest scop, limbajele de programare de nivel înalt (limbaje structurate) oferă structuri de control care servesc pentru a specifica ce trebuie să realizeze programul nostru, când și sub ce circumstanțe.

Un program de calculator conține de obicei trei structuri: secvențe de instrucțiuni, decizii și bucle. O secvență este un set de instrucțiuni care se execută secvențial. Decizia este o ramură (salt) din cadrul unui program bazat pe o anumită condiție. O buclă este o secvență de instrucțiuni care va fi executată în mod repetat pe baza unor condiții. În acest capitol vom explora unele dintre structurile de decizie comune într-un limbaj de asamblare [9].

5.1.1. Decizia simplă și cea compusă

Decizia simplă (`IF`) este utilizată pentru a executa o declarație sau un bloc de instrucțiuni numai dacă o condiție este îndeplinită. Forma sa este (în sintaxă C-like):

```
if (condiție)
  {
       // ramura IF
  }
```

unde *condiție* este expresia care trebuie evaluată. Dacă această condiție este adevărată, declarația sau blocul de instrucțiuni este executată. Dacă este falsă, declarația este ignorată (nu este executată) și programul continuă imediat după aceasta.

Condiția simplă se implementează în asamblare după șablonul (pseudo-cod):

```
Evaluează condiție
```

```
        Salt condiționat Jcc (pe condiție falsă) la eticheta_1
                ; ramura_IF
eticheta_1:
```

În plus, se poate specifica ce se dorește să se întâmple dacă condiția nu este îndeplinită, se obține astfel decizia compusă IF-ELSE. Forma sa este (C-like):

```
if (condiție)
  {
      // ramura IF
  }
else
  {
      // ramura ELSE
  }
```

și se implementează în asamblare după șablonul (pseudo-cod):

```
        Evaluează condiție
        Salt condiționat Jcc (pe condiție falsă) la eticheta_1
              ; ramura IF
        jmp eticheta_2
eticheta_1:
              ; ramura ELSE
eticheta_2:
```

Evaluarea condițiilor logice simple se realizează prin instrucțiuni de comparație, aritmetice etc., care poziționează bistabilii de condiție. De exemplu, secvența (C-like):

```
if (rax < ebx)
      // ramura IF
else
      // ramura ELSE
```

se implementează prin (dacă considerăm valorile din RAX și RBX cu semn) în asamblare:

```
        cmp rax, rbx
        jge et1
              ; ramura IF
et1:
              ; ramura ELSE
```

Evaluarea condițiilor complexe se abordează în manieră ordonată. Primul caz de bază este cel în care sub-condițiile sunt conectate prin operatorul logic AND (ȘI logic). să considerăm o condiție logică de forma:

$$C = C_1 \ AND \ C_2 \ AND \ C_3 \ AND \ ... \ AND \ C_n$$

și decizia compusă (C-like):

```
if (C)
      // ramura IF
else
      // ramura ELSE
```

implementarea este următoarea (pseudo-cod):

```
        Evaluează condiția C1
```

```
        Salt condiționat Jcc (pe condiția C1 falsă) la et_1
        Evaluează condiția C2
        Salt condiționat Jcc (pe condiția C2 falsă) la et_1
                ...
        Evaluează condiția Cn
        Salt condiționat Jcc (pe condiția Cn falsă) la et_1
                ; ramura_IF
        jmp et_2
et_1:
                ; ramura_ELSE
et_2:
```

De exemplu, dacă considerăm următoarea secvența de cod (C-like):

```
unsigned char chr;
unsigned long long i;
if (chr >= '0' && chr <= '9')
  {
       sir[i] = chr - '0';
       i++;
  }
```

în care se va încărca valoarea variabilei `chr` în registrul `AL` și a pointerului la șir în `RBX`:

```
    ...
    mov     al, chr
    lea     rbx, sir
    cmp     al, '0'
    jb      et_1
    cmp     al, '9'
    ja      et_1
    sub     al, '0'
    mov     [rbx], al
    inc     rbx
et_1:
    ...
```

Un al caz este cel în care condițiile sunt conectate prin operatorul `OR` (`SAU` logic), Să considerăm condiția logică de forma:

```
C = C1 OR C2 OR C3 OR ... OR Cn
```

și decizia compusă (C-like):

```
if (C)
        // ramura IF
else
        // ramura ELSE
```

implementarea rezultată este de forma (în pseudo-cod):

```
Evaluează condiția C1
Salt condiționat Jcc (pe condiția C1 adevărată) la et_1
Evaluează condiția C2
Salt condiționat Jcc (pe condiția C2 adevărată) la et_1
        ...
```

```
        Evaluează condiția Cn
        Salt condiționat Jcc (pe condiția Cn adevărată) la et_1
              ; ramura_ELSE
        jmp et_2
et_1:
              ; ramura_IF
et_2:
```

Cele două șabloane de implementare pentru condiții compuse de tip AND și OR se bazează pe proprietățile elementare ale operațiilor logice respective. Astfel, la operația AND, e suficient ca un singur operand (o sub-condiție) să fie falsă, pentru ca întreaga condiție să fie falsă. Similar, la operația OR, e suficient ca un singur operand să fie adevărat, pentru ca întreaga condiție să fie adevărată.

Cea de a treia schemă de dezvoltare se referă la implementarea condițiilor negate. O secvență de forma (C-like):

```
if (NOT condiție)
  {
      // ramura IF
  }
else
  {
      // ramura ELSE
  }
```

se implementează pe baza modelului de la schema IF-ELSE cu diferența că saltul condiționat se inversează (modelul rezultat se poate observa mai jos).

```
        Evaluează condiție
        Salt condiționat Jcc (pe condiție adevărată) la et_1
              ; ramura IF
        jmp et_2
et_1:
              ; ramura ELSE
et_2:
```

Pe baza celor trei operații logice elementare (AND, OR și NOT) se poate evalua orice tip de condiție logică. Dacă considerăm următoarea secvență (C-like):

```
if ((C1 AND C2) OR C3)
      // ramura IF
else
      // ramura ELSE
```

În prima etapă se evaluează expresiile după șablonul OR (expresiile (C1 AND C2) și C3):

```
        Evaluează condiția (C1 AND C2)
        Salt condiționat Jcc (pe condiția (C1 AND C2) adevărată) la et_1
et_3:
        Evaluează condiția C3
        Salt condiționat Jcc (pe condiție C2 adevărată) la et_1
              ...
        Evaluează condiția Cn
        Salt condiționat Jcc (pe condiție Cn adevărată) la et_1
```

```
            ; ramura_ELSE
      jmp et_2
et_1:
            ; ramura_IF
et_2:
```

În următoarea etapă se descompune condiția (`C1 AND C2`) conform șablonului pentru `OR`. Se observă că se va sări la eticheta `et_1` dacă ambele condiții sunt adevărate, altfel se va sări la eticheta `et_3`.

```
      Evaluează condiția C1
      Salt condiționat Jcc (pe condiția C1 falsă) la et_3
      Evaluează condiția C2
      Salt condiționat Jcc (pe condiția C2 adevărată) la et_1
et_3:
      Evaluează condiția C3
      Salt condiționat Jcc (pe condiție C2 adevărată) la et_1
            ; ramura_ELSE
      jmp et_2
et_1:
      ; ramura_IF
et_2:
```

Pe baza acestor modele aceste modele se pot evalua pas cu pas orice condiție logică. Aceste șabloane se utilizează de asemeni și pentru celelalte structuri din programarea structurată.

5.1.2. Selecția

Operația de selecție se descrie prin:

```
switch( C ) {
      case C1:
            // bloc SWITCH_C1
      [break;]     // optional
      case C2:
            // bloc SWITCH_C2
      [break;]     // optional
      ...
      case Cn:
            // bloc SWITCH_Cn
      [break;]     // optional
      [default:]   // optional
            // bloc SWITCH_DEFAULT
}
```

unde *C1*, *C2*, ... *Cn* sunt expresii constante, așa numitele cazuri (`case`). Acestea sunt, de fapt, valori de același tip cu variabila *C*. Cazul `default` corespunde situației în care variabila *C* nu are nici una din valorile *C1*, *C2*, ... *Cn* și este opțional. Blocurile `SWITCH_C1`, `SWITCH_C2` ... `SWITCH_Cn` pot fi și vide.

Implementarea naturală a selecției pornește de la observația că selecția este echivalentă cu o succesiune de decizii, după cum urmează:

```
if( C = C1 ) {
            // bloc SWITCH_C1
```

```
        }else if( C = C1 ) {
                        // bloc SWITCH_C2
              }else if ...
                        ...
                              else if( C = Cn ) {
                                          // bloc SWITCH_C2
                                    }else {
                                          // bloc SWITCH_DEFAULT
                                    }
```

Se poate aplica astfel modelul de la operația de decizie (comparații succesive și salturi condiționate), dar această soluție devine incomodă pentru un număr mare de cazuri:

```
        Evaluează condiție C=C1
        Salt condiționat Jcc (pe condiție adevărată) la et_1
        Evaluează condiție C=C1
        Salt condiționat Jcc (pe condiție adevărată) la et_2
              ...
        Evaluează condiție C=CN
        Salt condiționat Jcc (pe condiție adevărată) la et_n
              ; bloc SWITCH_DEFAULT
        jmp et_end
et_1:
              ; bloc SWITCH_C1
        jmp et_end
et_2:
              ; bloc SWITCH_C2
        jmp et_end
              ...
et_n:
              ; bloc SWITCH_Cn
        jmp et_end
et_end:  ; ieșirea din operație
```

O soluție de implementare, mult mai eficientă constă în utilizarea tabelelor de salt, inițializate cu punctele de intrare în blocurile de instrucțiuni (adresele etichetelor `et_i`, `i=1,n`) și calculul automat al adresei corespunzătoare de salt, pe baza căutării valorii curente C într-un tabel de cazuri posibile C_1, C_2, ... C_n.

5.1.3. Bucle cu test anterior

Buclele cu test anterior, descrise de secvența (C-like):

```
        while (condiție)
          {
              // bloc WHILE
          }
```

se implementează după modelul (pseudo-cod):

```
et_1:
        Evaluează condiție
        Salt condiționat Jcc (pe condiție falsă) la et_2
              ; bucla WHILE
```

```
        jmp et_1
    et_2:
```

Practic se evaluează condiția de ieșire din buclă (inversul contiției de „rămânere” în buclă). Dacă considerăm următoarea secvență de program (C-like):

```
    n = 0;
    while (*sir >= '0' && *sir <='9')
      {
          n = 10 * n + *sir - '0'
          sir++;
      }
```

unde `sir` este un pointer la un sir de caractere, iar `n` este un întreg pe 64 biți. Această secvență de cod este tipică pentru conversia ASCII-întreg. Dacă variabila `n` se va găsi în registrul `RAX`, iar pointerul `sir` în registrul `RSI` vom obține secvența (pseudo-cod):

```
        xor rax, rax
        lea rsi, sir
    et_1:
        Evaluează condiția ( [RSI] >= '0')
        Salt condiționat Jcc (pe condiție falsă) la et_2
        Evaluează condiția ( [RSI] <= '9')
        Salt condiționat Jcc (pe condiție falsă) la et_2
            RAX = RAX * 10 + [RSI] - '0'
            RSI++
        jmp et_1
    et_2:
        mov n, rax
```

Secvența propriu-zisă în asamblare care se obține este (în care conținutul registrului `[RSI]` va fi încărcat în registrul `AL` pentru o operare mai eficientă):

```
        lea    rsi, sir            ; incarcare adresa sir in RSI
        xor    ax, ax              ; AX = 0 (n)
        mov    bx, 10              ; BX = 10
    et_1:
        movzx  cx, byte ptr [rsi]  ; CX = caracterul de la adresa RSI (*sir)
        cmp    cl, '0'             ; evaluare prima sub-conditie
        jb     et_2                ; salt pe conditie falsa
        cmp    cl, '9'             ; evaluare a doua sub-conditie
        ja     et_2                ; salt pe conditie falsa
        mul    bx                  ; DX:AX = AX * BX (n = n * 10)
        sub    cl, '0'             ; CL =- '0' (conversie caracter - cifra)
        add    ax, cx              ; AX += CL (n = n + [RSI] - '0')
        inc    rsi                 ; RSI++ (*sir++)
        jmp    et_1
    et_2:
        mov n, rax                 ; n = RAX = n * 10 + *sir - '0'
```

5.1.4. Bucle cu număr cunoscut de pași

Bucle cu număr cunoscut de pași (sau buclele `FOR`) sunt un caz particular de bucle cu test anterior (`WHILE`) cu diferența că, condiția după care se efectuează bucla se bazează pe un contor

numeric care nu este modificat în corpul buclei ci în însăși structura ei (la finalul acesteia).

```
for (inițializare, condiții, actualizări)
{
      // bloc FOR
}
```

Această descriere se poate detalia pe baza unui ciclu `WHILE`:

```
inițializare();
while (contiții)
{
      // bloc FOR
  actualizări();
}
```

De exemplu dacă considerăm următoarea secvență din limbajul C:

```
for (i=0, i<100, i++)
{
      // bloc FOR
}
```

O implementare în asamblare poate fi:

```
   xor    edi, edi            ; EDI = 0 (inițializare contor)
et_1:
   cmp    edi, 100            ; comparatie EDI (contor) cu 100
   jnl    et_2                ; salt pe conditia falsă !(<100)
          ; bloc FOR => corpul buclei FOR (a nu se modifica contorul EDI)
   inc    edi                 ; EDI++ (actualizare contor)
   jmp    et_1                ; salt la începutul buclei
et_2:
```

5.1.5. Bucle cu test posterior

Buclele cu test posterior, descris prin una din formele:

```
do {                                repeat {
      // bloc DO-WHILE                        // bloc REPEAT-UNTIL
}while(conditie)                    }until(conditie)
```

se implementează după modelul pentru bucla `DO-WHILE`:

```
et:
      ; bucla DO-WHILE
   Evaluează condiție
   Salt condiționat Jcc (pe condiție falsă) la et
```

respectiv pentru `REPEAT-UNTIL`:

```
et:
      ; bucla REPEAT-UNTIL
   Evaluează condiție
   Salt condiționat Jcc (pe condiție adevărată) la et
```

Fie un algoritm tipic pentru conversia întreg-ASCII, descris în limbajul C:

```
        do {
                *sir++ = n % 10 + '0'
                n = n / 10;
        } while(n != 0)
        *sir = '\0'
```

Prin împărțiri succesive la 10 se generează cifrele corespunzătoare întregului `n` și se depun în șirul de caractere `sir` (cifrele rezultă în ordine inversă). Adresa șirului `sir` va fi încărcată în registrul `RDI`, iar numărul `n` în registrul `AX`. Codul rezultat este:

```
        lea     rdi, sir            ; incarcare adresa sir in RDI
        mov     ax, n               ; AX = n
        mov     bx, 10              ; BX = 10
    et:
        xor     dx, dx              ; DX = 0 (deimpartitul este în DX:AX)
        div     bx                  ; DX:AX / BX => AX=catul, DX = restul
        add     dl, '0'             ; DL += '0' = n % 10 + '0'
        mov     byte ptr [rdi], dl  ; depunere DL în *sir
        inc     rdi                 ; RDI++ (*sir++)
        test    ax, ax              ; comparatie ax cu 0
        jnz     et                  ; reluare bucla daca !=0
        mov     byte ptr [rdi], 0   ; *sir = '\0'
```

Introducere

Limbajul natural este limbajul care este utilizat de ființele umane pentru vorbire, scriere sau comunicare prin semne, care se diferențiază de limbajele formale, cum sunt limbajele de programare sau limbajele utilizate în studiul logicii formale în particular logica matematică. Limbajul natural are diverse forme, precum vorbirea, limbajul semnelor sau scrierea. Astfel, se deosebește de limbajul artificial și formal.

În matematică, logică și informatică, un *limbaj formal* este o mulțime de cuvinte de lungime finită (șiruri de caractere) bazate pe un alfabet finit.

Programarea este dispunerea cronologică a unor mișcări, operații, acțiuni sau activități astfel încât în finalul perioadei să se realizeze o stare posibilă a unui sistem.

Un *limbaj de programare* este un set bine definit de expresii și reguli (sau tehnici) valide de formulare a instrucțiunilor pentru un computer. Un limbaj de programare are definite un set de reguli sintactice și semantice. El dă posibilitatea programatorului să specifice în mod exact și amănunțit acțiunile pe care trebuie să le execute calculatorul, în ce ordine și cu ce date. Specificarea constă practic în întocmirea/scrierea programelor necesare („programare”).

Programarea informatică este o activitate informatică de elaborare a produselor-program, a programelor (software) necesare activităților realizate cu ajutorul calculatorului. Programarea informatică conține următoarele sub-activități: specificarea, proiectarea, implementarea, verificarea și testarea, documentarea și întreținerea produsului program.

Un *program* este un set de instrucțiuni care respectă regulile limbajului de programare ales.

Primele limbaje de programare predomină calculatorul modern. Pe parcursul unei perioade de nouă luni, între anii 1842 și 1843, Ada Lovelace a tradus memoriile matematicianul italian Luigi Menabrea cu privire la mașina de calcul propusă de Charles Babbage numită *Analytical Engine*. Aceasta a anexat articolului un set de note care specificau în detaliu o metodă de calcul pentru numerele Bernoulli folosind mașina de calcul a lui Babbage, recunoscută de unii istorici drept primul program. Unii biografi contestă totuși dimensiunea contribuțiilor ei originale față de cele ale soțului ei.

Limbajul mașină este o colecție de cifre binare sau biți pe care mașina de calcul le citește și interpretează. Limbajul mașină este singura limbă „înțeleasă” de calculatoare.

Un program scris în *limbaj de asamblare* constă într-o serie de instrucțiuni denumite mnemonice și care corespund unei secvențe de instrucțiuni executabile, când acesta este tradus în cod mașină de către un *asamblor*. Codul astfel obținut poate fi încărcat în memorie și executat. Limbajele de asamblare utilizează cuvinte cheie și simboluri, similare limbii engleze, pentru a forma un limbaj de programare.

În anii 1940 au apărut primele calculatoare electrice moderne. Viteza de calcul și capacitatea de memorare limitată au forțat programatorii să scrie programe în limbaj de asamblare. S-a constatat astfel că programarea în limbaj de asamblare necesită mult efort intelectual și este predispus la erori.

Limbajele de programare de nivel înalt (HLP[1]) permit scrierea de programe folosind instrucțiuni asemănătoare cu cuvintele din limbajul natural scris care sunt apoi traduse în limbaj mașină pentru a fi executate.

Anii 1960 și 1970 se remarcă din punct de vedere a dezvoltării limbajelor de programare prin polemica pe baza meritelor „programării structurate”, care în esență înseamnă programarea fără utilizarea instrucțiunilor de salt (`GOTO`).

1 HLP – din engl. High-Level Programing

Anii 1980 au fost ani de consolidare și maturizare a limbajelor de programare. C++ combină programarea orientate pe obiecte și programarea structurată. O nouă tendință importantă în designul limbajelor este dată de o concentrare sporită asupra programării la scară largă prin utilizarea modulelor sau a unităților organizaționale de dimensiuni mari la nivel de cod.

Limbajele de programare se împart în 5 generații:

- generația I: limbajele de nivel jos care sunt limbajele mașină;
- generația a II-a: de-asemeni limbaje de nivel jos de tip limbaje de asamblare;
- generația a III-a: limbaje de nivel înalt (precum limbajul C);
- generația a IV-a: sunt limbaje care constau în declarații similare limbii naturale. Limbajele celei de-a patra generații sunt utilizate în mod frecvent în programarea bazelor de date și în scripturi;
- generația a V-a: sunt limbaje de programare care conțin instrumente vizuale ce ajută la dezvoltarea programelor (un bun exemplu este Visual C).

Tabelul 1 conține o succintă comparație între cele două tipuri de limbaje nivel jos și limbajele de nivel înalt.

Tabelul 1. Comparație între tipuri de limbaje de programare.

	Limbajele mașină	*Limbaje de asamblare*	*Limbaje de nivel înalt*
Timpul de execuție	Deoarece este limbajul de bază al calculatoarelor, nu necesită traducere, deci asigură eficiența mașinii. Programele rulează mai repede.	Este necesar un *asamblor* pentru a converti programul în limbajul mașinii.	Un program numit *compilator* sau *interpretor* este necesar pentru a converti programul în limbaj mașină. Astfel, este nevoie de mai mult timp pentru execuție.
Dezvoltarea aplicației	Necesită pricepere și experientă deoarece secvențele de instrucțiuni sunt lungi și complexe. Este nevoie de mai mult timp pentru a programa. Sunt dificil de depanat și întreținut.	Mai ușor de utilizat decât limbajul mașinii, dar necesită memorarea codurilor instrucțiunilor. Este nevoie de mai puțin timp pentru a se dezvolta programe comparativ cu limbajele mașină.	Ușor de utilizat. Ia mai puțin timp pentru dezvoltarea de programe și, prin urmare, asigură o eficiență sporită a programului.

Capitolul 1. Reprezentarea datelor

Definiție: Un *sistem de numerație* constituie totalitatea regulilor de reprezentare a numerelor cu ajutorul anumitor simboluri, denumite cifre (engl. *Digit*). Oamenii sunt obișnuiți cu sistemul de numerație zecimal, probabil deoarece oamenii au 10 degete. Cu toate acestea, numărul „10” nu are semnificații semnificative în știință și matematică. Sistemul de numerație natural în electronica digitală este cel binar: „`0`” corespunde absenței curentului electric și „`1`” prezenței acestuia.

Definiție: Un sistem de numerație se numește *pozițional*, dacă ponderea unei cifre este dată de poziția pe care aceasta o ocupă în cadrul numărului. De exemplu dacă considerăm numărul `2019` scris în baza 10, `9` este numărul de unități, `1` numărul zecilor, `0` al sutelor și `2` al miilor.

Conform convenției numerelor poziționale aceste se scriu de la stânga la dreapta începând cu cifra cea mai semnificativă și terminând cu cea mai nesemnificativă.

Definiție: Unitatea de bază de memorare a informație se numește *bit* (prescurtare de la *Binary Digit*, în traducere cifră binară). Noțiunea de bit a fost utilizată pentru prima dată în teza de doctorat a matematicianului Claude Shannon, care prin teza sa a deschis drumul către un nou domeniu numit teoria informației.

Definiție: Se numește *baza sistemului de numerație* numărul total de cifre distincte utilizate într-un sistem de numerație. Dacă notăm baza sistemului de numerație cu *b* aceasta trebuie să satisfacă condiția: $b>1$. Numerele pot fi reprezentate în baza *b* folosindu-se cifrele cuprinse în intervalul $[0, b-1]$.

În anul 1964 proiectanții calculatorului IBM System/360 au stabilit ca și convenție folosirea grupurilor de 8 biți ca unitate de bază a memoriei calculatorului. Astfel a apărut *octetul* sau *byte*-ul.

Un *cuvânt* este format din doi sau mai mulți octeți adiacenți adresați și manipulați împreună. Mărimea cuvântului reprezintă mărimea datelor care sunt optim manevrate de către o anumită arhitectură. Cuvintele pot fi succesiuni de 16, 32, 64 biți. O succesiune de 4 biți (jumătate de octet) se numește *nibble*.

1.1. Baze de numerație

Conform convenției numerelor poziționale, aceste se scriu de la stânga la dreapta începând cu cifra cea mai semnificativă și terminând cu cea mai nesemnificativă. Astfel pentru un număr întreg de `n` cifre numerotate de la `n-1` până la `0`, cifra cea mai din stânga va fi cea mai semnificativă și va avea rangul `n-1`, iar cifra cea mai din dreapta este cea mai nesemnificativă și va avea rangul `0`.

Fie un număr scris într-o bază oarecare *b* sub forma parte întreagă și parte zecimală (fracționară):

$$Nr_{(b)} = I_n I_{n-1} I_{n-2} \ldots I_2 I_1 I_0 , F_1 F_2 F_3 \ldots \quad , \tag{1}$$

atunci valoarea sa în baza 10 va fi:

$$\begin{aligned} Nr_{(10)} = I_n * b^n + I_{n-1} * b^{n-1} + \ldots + I_2 * b^2 + I_1 * b^1 + I_0 * b^0 + \\ + F_1 * b^{-1} + F_2 * b^{-2} + F_3 * b^{-3} + \ldots \end{aligned} \quad . \tag{2}$$

1.1.1. Conversia numerelor din baza 10 într-o bază oarecare

Pentru conversia unui număr care conține atât parte întreagă cât și parte zecimală trebuie convertite separat partea întreagă și cea zecimală.

Pentru conversia părții întregi algoritmul cel mai simplu constă în împărțirea succesivă a numărului scris în baza 10 la baza spre care se dorește conversia (se împarte numărul la bază, iar în continuare se împarte câtul obținut la bază ș.a.m.d. până când câtul devine `0`), după care se iau resturile obținute în ordine inversă, care constituie valoarea numărului în baza cerută.

Pentru conversia numerelor între bazele 2, 8 și 16 există o metodă mai rapidă. Dacă se ține cont de faptul că pentru fiecare cifră hexazecimală există 4 cifre binare corespondente iar pentru fiecare cifră în octal există 3 cifre binare se obține valorile din Tabelului 1.1.

Tabelul 1.1. Conversia între bazele 10, 16, 2, respectiv 8.

Zecimal	*Hexazecimal*	*Binar*	*Octal*
0	0	0000	000
1	1	0001	001
2	2	0010	010
3	3	0011	011
4	4	0100	100
5	5	0101	101
6	6	0110	110
7	7	0111	111
8	8	1000	
9	9	1001	
10	A	1010	
11	B	1011	
12	C	1100	
13	D	1101	
14	E	1110	
15	F	1111	

Pentru a converti un număr subunitar (deci partea fracționară a unui număr) din baza 10 într-o bază oarecare se fac înmulțiri succesive ale părților fracționare până când se ajunge la parte fracționară nulă, sau se ajunge la perioadă sau se depășește capacitatea de reprezentare (se obțin cifre suficiente, deși algoritmul nu s-ar fi terminat). Ceea ce depășește partea zecimală la fiecare înmulțire reprezintă o cifră a numărului în baza spre care se face conversia.

Conversia unui număr care are atât parte întreagă cât și parte zecimală se face convertind pe rând partea întreagă și cea zecimală.

1.1.2. Conversia unui număr dintr-o bază oarecare în baza 10

Pentru a converti un număr scris conform ecuației (1) dintr-o bază oarecare în baza 10 se poate folosi direct formula (2).

1.1.3. Sistemul de numerație binar

Majoritatea sistemelor informatice moderne funcționează folosind logica binară. Calculatorul reprezintă valorile folosind două niveluri de tensiune (de exemplu 0 V și +5 V). Cu două astfel de niveluri se pot reprezenta exact două valori diferite. Acestea pot fi două valori diferite, dar prin convenție folosim valorile `0` și `1`. Aceste două valori, întâmplător, corespund celor două cifre folosite de sistemul de numerație binar. Întrucât există o corespondență între nivelurile logice utilizate de sistemele x86 și cele două cifre utilizate în sistemul de numerație binar, sistemele IBM-PC folosesc sistemul binar.

Sistemul binare funcționează la fel ca sistemul de numerație zecimal, cu două excepții: sistemul binar permite doar cifrele `0` și `1` (mai degrabă decât `0`...`9`), iar binarul folosește puteri ale lui 2, mai degrabă decât puteri ale lui 10. Prin urmare conversia, unui număr binar în zecimal este foarte ușoară. Pentru fiecare cifră de „1” din șirul binar, se adună 2^n unde „n” este convențional poziția de la stânga la dreapta a cifrei binare. De exemplu, valoarea binară $10101011_{(2)}$ se reprezintă ca:

$$1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+1*2^1+1*2^0=128+32+8+2+1=171_{(10)}$$

Conversia unui număr zecimal în binar este puțin mai dificilă. Fie se obține numărul prin împărțiri repetate la 2, iar resturile obținute până când câtul devine `0` reprezintă numărul, fie se pot căuta puteri ale lui 2 care sumate împreună dau valoarea zecimală dorită. De exemplu valoarea binară a numărului `579` se poate obține urmând pașii:

- Dacă $2^{10}=1024$, $2^9=512$ este cea mai mare putere a alui 2 mai mică decât numărul `579`. Valoarea binară va începe cu „1” (cifra binară cea mai semnificativă). Se va calcula diferența dintre numărul `579` și puterea lui $2^9=512$ (`579-512=67`).
- Următoarea putere a lui 2, $2^8=256$ este mai mare decât rezultatul anterior și prin urmare se va adăuga o cifră de `0` valorii binare („10”).
- $2^7=128$ este mai mare decât rezultatul anterior, astfel valoarea binară devine „100”.
- Următoare putere a lui 2, $2^6=64$ este mai mică decât valoarea anterioară, deci valoarea binară devine „1001” și valoarea rămasă va fi `67-64=3.`
- Puterile lui 2, $2^5=32$, $2^4=16$, $2^3=8$ și $2^2=4$ sunt mai mari decât restul rămas astfel valoarea binară rezultată va fi „10010000”.
- $2^1=2$ este mai mică decât valoarea anterioară `3`, astfel valoarea binară va fi „100100001” și restul `3-2=1`.
- Ultima valoare binară $2^0=1$ este egală cu valoarea rămasă astfel încât valorii binare i se va adăuga o cifră de „1”. Valoare binară finală este „1001000011”.

Teoretic un număr binar conține un număr infinit de cifre (sau biți). De exemplu, numărul $9_{(10)}$ (nouă în baza 10) poate fi scris ca: $1001_{(2)}$, $01001_{(2)}$ sau $000000000001001_{(2)}$, etc.. Orice număr de biți de zero pot preceda un număr binar fără a-i modifica valoarea.

Prin convenție, în exprimarea valorilor numerice se ignoră cifrele de `0` ce preced numerele. De exemplu, $1001_{(2)}$ reprezintă numărul zecimal 9. În programare în schimb, cifrele de `0` ce preced valorile numerice au o semnificație anume. Acestea specifică numărul de biți pe care este reprezentată acea valoare. Astfel, numărul zecimal 9 reprezentat, de exemplu, pe 8 biți este: $00001001_{(2)}$ iar pe 32 biți este: $00000000000000000000000000001001_{(2)}$.

1.1.4. Sistemul de numerație hexazecimal

O problemă a numerelor reprezentate în sistemului binar este faptul că acestea au multe cifre. Sistemul zecimal reprezintă numerele mult mai compact decât sistemul binar. Din păcate,

sistemele de calcul operează în mod binar, de aceea de cele mai multe ori este convenabilă utilizarea sistemul de numerotare binar. Deși sistemul zecimal este mai compact conversia din baza 2 în baza 10 și invers nu este o operație banală. Sistemul de numerotare hexazecimal (baza 16) rezolvă aceste probleme. Din faptul că 16 este o putere a lui 2 ($2^4=16$) rezultă faptul că o cifră hexazecimală se reprezintă în binar pe *exact* 4 cifre binare. Reprezentarea numerelor în hexazecimal oferă astfel două avantaje: sunt foarte compacte (comparativ cu reprezentarea binară), iar conversia din binar și invers este foarte simplă. Din această cauză, majoritatea sistemelor informatice binare folosesc sistemul de reprezentare hexazecimal, mai ales pentru numere reprezentate pe un număr relativ mare de biți (cum ar fi de exemplu adresele de memorie).

Fiecare cifră hexazecimală poate reprezenta una dintre cele șaisprezece valori cuprinse între `0` și $15_{(10)}$. Deoarece există doar zece cifre zecimale, numerele reprezentate în hexazecimal necesită șase cifre suplimentare pentru a reprezenta valorile din intervalul de la $10_{(10)}$ până la $15_{(10)}$. Pentru reprezentarea acestor valori se utilizează literele de la `A` până la `F` (a se vedea primele două coloane din Tabelul 1.1).

Pentru a converti un număr hexazecimal într-un număr binar, se înlocuiește fiecare cifră hexazecimală a numărului cu combinația binară pe patru biți corespunzătoare din Tabelul 1.1. De exemplu valoarea binară a numărului hexazecimal $A1B2_{(16)}$ este $1010\ 0001\ 1011\ 0010_{(2)}$.

Conversia unui număr binar în format hexazecimal este este fel de ușoară. Numărul binar se împarte în grupe de câte patru biți, începând de la bitul cel mai nesemnificativ (conform convenției de scriere: de la dreapta la stânga) și se înlocuiesc cu cifra hexazecimală corespunzătoare. Dacă este necesar ultima grupare de biți (cei mai semnificativi) se for completă cu zerouri.

Se poate observa astfel un alt avantaj al folosirii numerelor hexazecimale, și anume faptul că se poate identifica valoarea unui bit (sau a mai multor biți) fără a converti integral numărul în binar. Aceste conversii parțiale sunt posibile datorită faptului că o cifră hexazecimală ocupă *exact* 4 biți, astfel încât valoarea unui bit depinde doar de cifra din care face parte (la conversie nu există transport între valorile cifrelor hexa alăturate).

1.2. Reprezentări interne ale datelor

Marea majoritate a sistemelor de calcul moderne folosesc sistemul binar. De aici rezultă faptul esențial că, în memoria sistemelor de calcul, informația de orice fel (date sau program) este întotdeauna reprezentată în formă binară.

Informația este organizată în sistemele de calcul în grupe de câte 8 biți (numit octet sau *byte*). Octetul este unitatea de măsură în care se exprimă volumul memoriei unui sistem de calcul.

Pentru reprezentarea unităților de memorie (octet sau *byte*) și unității de informație (bit) se pot utiliza multiplii acestora. Deoarece sistemele de calcul utilizează sistemul binar, pentru reprezentarea cantității de date, în domeniul informatic, se utilizează multiplii binari în locul celor zecimali. Dacă multiplii zecimali sunt puteri (multiplu de 3) ai bazei de numerație zecimal (10), multiplii binari sunt puteri (multiplu de 10) ai bazei de numerație binară (2).

În decembrie 1998, Comisia electrotehnică internațională (IEC[1]) a aprobat ca standard internațional denumiri și simboluri de prefix-uri pentru multipli binari pentru utilizarea în domeniile prelucrării și transmiterii datelor. Noile prefixe pentru multipli binari nu fac parte din

1 IEC – International Electrotechnical Commission este o organizație internațională de standardizare care pregătește și publică standarde internaționale pentru toate tehnologiile electrice, electronice și conexe – cunoscute colectiv ca „electro-tehnologie”

Sistemul internațional de unități (SI[1]). Cu toate acestea, pentru ușurința în înțelegere și memorare, acestea au fost obținute din prefixele SI pentru puteri pozitive ale lui zece. După cum se poate observa din Tabelul 1.2, numele fiecărui nou prefix este derivat din numele prefixului SI corespunzător prin păstrarea primelor două litere ale numelui prefixului SI și adăugarea literelor „bi", de la cuvântul „binar". În mod similar, simbolul fiecărui prefix nou este derivat din simbolul prefixului SI corespunzător prin adăugarea literei „i", care amintește din nou de cuvântul „binar" (pentru consecvență cu celelalte prefixe pentru multipli binari, simbolul *Ki* este utilizat pentru 2^{10}, mai degrabă decât *ki*) [1].

Tabelul 1.2. Prefixe pentru multiplii metrici (SI) și binari (IEC).

Sistem metric (SI)			Sistemul binar (IEC)		
Valoare	Simbol	Unitate	Valoare	Simbol	Unitate
10^3	K	kilo	2^{10}	Ki	kibi
10^6	M	mega	2^{20}	Mi	mebi
10^9	G	giga	2^{30}	Gi	gibi
10^{12}	T	tera	2^{40}	Ti	tebi
10^{15}	P	peta	2^{50}	Pi	pebi
10^{18}	E	exa	2^{60}	Ei	exbi
10^{21}	Z	zetta	2^{70}	Zi	zebi
10^{24}	Y	yotta	2^{80}	Yi	yobi

La începuturile dezvoltării sistemelor de calcul, specialiștii în calculatoare au observat că 2^{10} era aproape egal cu 1000 și au început să folosească prefixul SI „kilo" pentru a desemna valoarea 1024. Asta a funcționat destul de bine timp de un deceniu sau doi, deoarece specialiștii în calculatoare care utilizau termenul *kilobytes* știau că termenul implică 1024 octeți (*bytes*). Însă odată cu răspândirea sistemelor de calcul au apărut „conflicte" în comunicarea dintre acești specialiștii și fizicienii, inginerii și chiar cu oamenii obișnuiți, pentru care un kilometru este de 1000 de metri și un kilogram este 1000 de grame.

Ulterior, au fost popularizați termeni precum *gigabyte* și chiar *terabyte* pentru stocarea datelor, deși pentru dispozitivele de stocare aritmetica binară era mai puțin convenabilă decât aritmetica zecimală. Rezultatul este că în prezent un termenul „megabyte", de exemplu, nu are aceeași însemnătate pentru „toată lumea". Pentru a specifica dimensiunea memoriei calculatoarelor, majoritatea producătorilor utilizează *megabyte* pentru a desemna valoarea 2^{20} = 1048576 octeți, în schimb pentru producătorii dispozitivelor de stocare pentru aceleași calculatoare, termenul însemnă 1000000 de octeți. Unii designeri de rețele de calculatoare au folosit *megabit pe secundă* pentru desemna valoarea de 1048576 biți/s, deși toți inginerii în telecomunicații o folosesc pentru a însemna 10^6 biți/s.

Ținând cont de toate acestea și pentru a elimina eventualele confuzii, Consiliul pentru standarde IEEE[2] (IEEE Standards Board) a decis prin standardul IEEE 1541-2002 utilizarea

1 Sistemul internațional de unități – în engl. International System of Units (SI) este forma modernă a sistemului metric și este cel mai utilizat sistem de măsurare.

2 IEEE (Institute of Electrical and Electronics Engineers) este o organizație internațională non-profit care are ca scop sprijinirea inovațiilor tehnologie din domeniul electric și electronic. IEEE a luat ființă în anul 1963 prin fuziunea dintre Institute of Radio Engineers (IRE), fondată în 1912 și American Institute of Electrical Engineers (AIEE), fondată în 1884.

multiplilor binari. În ciuda statutului oficial, unitățile binare nu sunt frecvent utilizate chiar și atunci când raportează numărul de octeți calculat în multipli binari. Discrepanța poate provoca confuzie, deoarece sistemele de operare care utilizează metoda binară raportează valori numerice mai mici pentru dimensiunea de stocare decât cele specificate de producători, cum ar fi producătorii de unități de stocare care utilizează strict unitățile zecimale.

Multe sisteme de operare calculează dimensiunea fișierelor în *mebibytes*, dar raportează numărul ca fiind MB (*megabytes*). De exemplu, toate versiunile sistemului de operare Microsoft Windows raportează un fișier de 2^{20} de octeți ca „1.00 MB” sau „1.024 KB” în fereastra de dialog cu proprietățile fișierului și raportează un fișier de 10^6 (1000000) octeți ca 976 KB. Mare parte din versiunile sistemelor de operare Linux utilizează, în schimb, multipli binari IEC pentru cantități binare și prefixe SI pentru cantități zecimale.

Unitatea de măsură pentru cantitatea de informație este indivizibilă și prin urmare, spre deosebire de unitățile de măsură fizice, nu au submultipli. Astfel, deși există subunitatea de măsură centimetru pentru lungime, *centibitul* sau *centioctetul* cu siguranță nu există!

1.2.1. Reprezentarea numerelor întregi fără semn

Numerele întregi pot fi reprezentate pe un număr oarecare de biți. Deoarece informația în sistemele de calcul este organizată pe octeți, numărul de biți pe care se pot reprezenta numerele este, de obicei, multiplu de 8 biți (1 octet). Astfel sunt uzuale numere întregi reprezentate pe 8, 16, 32, 64 sau 128 biți (1, 2, 4, 8 sau 16 octeți). Deoarece fiecare un bit poate lua doar două valori (`0` sau `1`), numărul total de valori distincte ce pot fi reprezentate pe `n` biți este 2^n.

Tabelul 1.3. Reprezentarea numerelor fără semn pe 4 biți.

Reprezentarea internă	Valoarea zecimală
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

În cazul numerelor fără semn, valoarea internă a biților reprezintă chiar valoarea numărului. Astfel domeniul de valori ce poate fi reprezentat pe `n` biți este de la `0` la `2n-1`. Tabelul 1.3 conține domeniul numerelor fără semn reprezentate pe `n=4` biți.

1.2.2. Reprezentarea numerelor întregi cu semn

Pentru reprezentarea numerelor cu semn valorile interne care se pot reprezenta (combinațiile binare) sunt aceleași ca la numerele fără semn. Se pune problema ca, printr-o convenție adecvată, să se considere o parte dintre aceste combinații binare ca reprezentând numere întregi pozitive, iar cealaltă parte numere negative.

Există mai multe sisteme de reprezentare a numerelor întregi cu semn dintre acestea, cele mai răspândite sunt: sistemul de reprezentare în complement față de 1, complement față de 2, cod Excess.

1.2.2.1. Reprezentarea în complement față de 1

În sistemul de reprezentare în complement față de 1, bitul cel mai semnificativ are un rol special, anume de a preciza semnul numărului (din acest motiv se numește *bit de semn*). Dacă bitul de semn este `0`, numărul este pozitiv, iar dacă acesta este `1`, numărul este negativ.

Tabelul 1.4. Reprezentarea numerelor cu semn pe 4 biți, în complement față de 1.

Reprezentarea internă (binar)	Valoarea fără semn	Valoarea în complement față de 1
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-7
1001	9	-6
1010	10	-5
1011	11	-4
1100	12	-3
1101	13	-2
1110	14	-1
1111	15	-0

Pentru numerele pozitive, reprezentarea în complement față de 1 (pe `n` biți) este identică cu reprezentarea numerelor binare cu semn pe `n-1` biți la care se adaugă un bit `0`, cel mai

semnificativ. În cazul numerelor negative, reprezentarea este obținută prin complementarea (negarea pe biți) a fiecărui bit (inclusiv bitul de semn). Complementarea unui număr pe `n` biți este echivalentă cu scăderea acestuia din valoarea 2^n-1. De exemplu pentru complementarea numărului pe 4 biți $0101_{(2)}$, `n=4` rezultă 2^4-1=15; scăzând valoarea $0101_{(2)}$ (5) din $1111_{(2)}$ (15) se obține valoarea $1010_{(2)}$ (-5). Tabelul 1.4 conține reprezentarea numerelor cu semn reprezentate pe `n=4` biți, în complement față de 1.

Se poate observa din Tabelul 1.4 că există două reprezentări ale lui zero: `0000` și `1111` în reprezentarea în complement față de 1, un `+0` și un `-0`.

Pentru un număr pe `n` biți, cel mai mare număr pozitiv reprezentabil în complement față de 1 este $2^{(n-1)}$-1, iar cel mai mic număr negativ care poate fi reprezentat este -($2^{(n-1)}$-1).

Operația de adunare a numerelor în complement față de 1 se efectuează în două etape (operația de scădere este similară – o scădere este echivalentă cu o sumă în care cel de-al doilea operant este complementat, *a-b=a+(-b)*):

- se efectuează suma reprezentărilor fără semn a numerelor (inclusiv bitul de semn);
- se adună bitul de depășire obținut de la pasul anterior.

Fie perechile de numere `6` și `-3` respectiv `+4` și `+3`, suma acestora este:

```
 0110+  (+6)          0100+  (+4)
 1100   (-3)          0011   (+3)
10010                 00111

 0010+                0111+
    1                    0
 0011   (+3)          0111   (+7)
```

Dezavantajele reprezentării în complement față de 1 sunt:

- existența a două reprezentări pentru valoarea zero (`+0` și `-0`) astfel încât, în cazul comparației cu zero, trebuie verificate ambele reprezentări;
- operațiile aritmetice implică două operații, prima etapă generează un transport care trebuie apoi sumat rezultatului intermediar.

Avantajele reprezentării în complementar față de 1 sunt:

- simplitatea calculului complementului față de 1 (pentru a obține valoarea negativată, biții trebuiesc doar inversați);
- pentru a converti un număr întreg cu semn la o dimensiune mai mare trebuie doar copiat bitul de semn în biții suplimentari (operație de extindere a semnului).

1.2.2.2. Reprezentarea în complement față de 2

Sistemul de reprezentare în complement față de 2 este cel mai răspândit sistem de reprezentare pentru numere întregi cu semn. Sistemele x86-64 folosesc această reprezentare.

Ca și sistemul de reprezentare în complement față de 1, sistemul în complement față de 2 folosește cel mai semnificativ bit din reprezentarea binară ca bit se semn.

Numerele pozitive, reprezentate în complement față de 2 sunt identice cu reprezentarea numerelor binare fără semn cu un bit de semn cu valoarea `0`. Reprezintarea numerelor negative se obține prin scăderea valorii pozitive din 2^n, unde `n` este numărul de biți pe care sunt reprezentate numerele. O metodă echivalentă, mai rapidă, pentru a obține complementul față de 2 se realizează în două etape: în prime etapă se inversează (complementează) valoarea fiecărui bit, după care la valoare obținută se adună valoarea `1`. O altă metodă contă în complementarea tuturor biților de după primul bit de 1 plecând la la cel mai nesemnificativ bit către cel mai

semnificativ.

Fie numărul 6, valoarea negată în complement față de 2 pentru n=4 biți este (folosint cele 3 metode prezentate anterior):

```
10000-  (2^n)        NOT(0110)=      -(0110)=
 0110   (6)             1001+          1010
 1010   (-6)              1
                       1010
```

Tabelul 1.5. Reprezentarea numerelor cu semn pe 4 biți, în complement față de 2.

Reprezentarea internă (binar)	Valoarea fără semn	Valoarea în complement față de 2
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

În Tabelul 1.5 se regăsesc reprezentările numerelor cu semn, pe n=4 biți, în complement față de 2. Se poate observa din Tabelul 1.5 că bitul cel mai semnificativ este bit de semn și în complement față de 2, dar există o singură reprezentare a lui zero (0). De asemenea se poate observa că reprezentarea în cod complementar față de 2 este asimetrică (pe 4 biți se poate reprezenta -8 da nu și +8).

Domeniul de reprezentare a numerelor întregi în complement față de 2 (pe n biți) este:

- 2^{n-1} valori pozitive de la 0 la $2^{n-1}-1$;
- 2^{n-1} valori negative de la -2^{n-1} la -1.

Adunarea în complement față de 2 este mai simplă ca în complement față de 1, deși obținerea numărului negativ este mai complicată. Adunarea și scăderea numerelor reprezentate în complement față de 2 este aceeași ca la numere fără semn (incluzând și bitul de semn), cu ignorarea depășirii (*carry*). Avantajul reprezentării în complement față de 2 este acela că,

adunând un număr cu complementul său față de 2 rezultatul obținut este 0 (ignorând depășirea) ceea ce nu este valabil în cazul celorlalte reprezentări.

Dezavantajele reprezentării în complement față de 2 sunt:

- operația de negare (complementare) este mai complicată față de cea pentru complement față de 1;
- reprezentarea în complement față de 2 este asimetrică.

Avantajele reprezentării în complement față de 2, datorită cărora aceasta s-a impus, sunt:

- o reprezentare unică pentru zero (0);
- simplitatea operațiilor aritmetice;
- conversia la o dimensiune mai mare se face printr-o simplă operație de extindere a semnului.

1.2.2.3. Reprezentarea în coduri Excess-N

Excess-N, uneori denumit și offset-N, este o schemă de reprezentare digitală în care combinația binară cu toți biții de 0 corespunde valorii negative minime și cea cu toți biții de 1 valorii maxime pozitive reprezentabile.

Schema de reprezentare Exces-N deplasează toate valorile cu N față de valoare lor binară. Adică, în reprezentare Excess-N, valoarea numărul se obține scăzând din N, valoarea fără semn (binară) a numărului. De exemplu, cod Excess-3, secvența „0000” (care are valoare 0 ca număr fără semn) reprezintă de fapt 0-3=-3. Secvența „0100” (care are valoare binară 4) are reprezentarea 4-3=1.

Termenul de *număr magic* se referă la o valoare deosebit de utilă pentru deplasamentul (N) în codurile Excess-N. Ideea de bază este de a deplasa numerele din intervalul reprezentabil, astfel încât jumătate dintre ele să fie pozitive și cealaltă jumătate să fie negative. Desigur, acest lucru nu este posibil, deoarece pe n biți, se pot reprezenta 2^n numere întregi diferite, iar valoarea 0 va fi una dintre acestea. Exceptând valoarea 0, rămân 2^n-1 combinații pentru a reprezenta numerele pozitive și negative. Deoarece 2^n-1 care este impar, nu se poate realiza o distribuție uniformă.

Dacă se alege 2^{n-1} ca valoare a deplasării, se vor putea reprezenta 2^{n-1} numere negative și $2^{n-1}-1$ numere întregi pozitive (inclusiv 0). Mai mult, numerele au cel mai semnificativ bit pe 0, în timp ce numerele întregi pozitive au bitul cel mai semnificativ pe 1. În acest sens, numerele în reprezentarea Excess-2^{n-1} se pot converti foarte ușor în complement față de 2 (și invers) prin inversarea valorii bitului de semn.

Similar se poate alege un deplasament de $2^{n-1}-1$. În acest caz gama de numere întregi care pot fi reprezentate va fi de la $-(2^{n-1}+1)$ până la $+(2^{n-1})$. Dintre acestea, numerele întregi cu bitul cel mai semnificativ cu valoarea 1 sunt pozitive, iar cele cu primul bit 0 sunt negative sau zero (0), astfel încât 0 este aliniat la numerele întregi negative. Pentru un deplasament de $2^{n-1}-1$ nu se mai poate face însă conversia în complement față de 2 prin simpla inversare a bitului de semn.

Cum se poate observa în Tabelul 1.6, deoarece ordinea numerelor în Excess-N este aceeași ca și valorile binare (reprezentarea fără semn), pentru o operație de comparație logică a numerelor în Excess-N, se obține același rezultat ca în cazul comparației numerice a reprezentării binare a numărului. În reprezentarea în complement față de 2, în schimb, o comparație logică va fi echivalentă cu o comparație a combinației binare interne doar pentru numere de același semn. În caz contrar, sensul comparației va fi inversat, deoarece toate valorile negative sunt în reprezentarea lor internă mai mari decât toate valorile pozitive.

Standardul IEEE pentru aritmetica în virgulă mobilă (IEEE 754) folosește diverse dimensiuni ale exponentului, care este reprezentat în cod Excess-$2^{n-1}-1$ (Excess-127, Excess-

1023, Excess-16383), ceea ce înseamnă că inversarea bitului de semn al exponentului nu va converti exponentul în complement față de 2.

Tabelul 1.6. Reprezentarea numerelor cu semn pe 4 biți, în cod Excess-7.

Reprezentarea internă (binar)	Valoarea fără semn	Valoarea în Excess-7
0000	0	-7
0001	1	-6
0010	2	-5
0011	3	-4
0100	4	-3
0101	5	-2
0110	6	-1
0111	7	0
1000	8	1
1001	9	2
1010	10	3
1011	11	4
1100	12	5
1101	13	6
1110	14	7
1111	15	8

Codurile Excess-N sunt adesea utilizate în procesarea digitală a semnalelor (DSP[1]). Majoritatea circuitelor de conversie analog–digital (A/D) și digital–analog (D/A) sunt unipolare, ceea ce înseamnă că nu pot procesa semnale bipolare (semnale cu valori pozitive și negative). O soluție simplă în acest sens este de a aplica semnalelor analogice un decalaj continuu egal cu jumătate din gama convertorului A/D și D/A. Datele digitale rezultate care se obțin vor fi implicit în reprezentare Excess-N.

Principalul avantaj al codurilor Excess-N rămâne simplitatea comparației valorilor numerice în această reprezentare.

1.2.3. Reprezentarea numerelor reale

Deși numerele întregi oferă o reprezentare exactă pentru valorile numerice, acestea suferă de două neajunsuri majore: incapacitatea de a reprezenta valori fracționare și un interval dinamic limitat. Numere reale rezolvă aceste două probleme în detrimentul preciziei și, pe unele procesoare, al vitezei.

Numere reale (fracționare) se pot reprezenta în două moduri distincte: în virgulă fixă respectiv virgulă mobilă.

Pentru multe aplicații, avantajele reprezentării în virgulă fixă/mobilă depășesc

1 DSP – din enlg. Digital Signal Processing.

dezavantajele. Cu toate acestea, pentru a utiliza în mod corespunzător aritmetica numerelor reale trebuie înțeleasă modalitatea de funcționare a acesteia. Intel, care a înțeles importanța aritmeticii în virgulă mobilă, a oferit suport pentru aritmetica în virgulă mobilă începând de la primele modele ale procesorului 8086 prin circuitul opțional lansat în 1980 sub numele de cod 80x87 (unitate în virgulă mobilă sau co-procesor matematic). Ulterior suportul pentru virgulă mobilă a fost extins prin setul de instrucțiuni 3DNow! lansate de AMD în 1998 și SSE lansate la scurt timp de Intel în 1999.

O mare problemă a aritmeticii cu numere reale este că aceasta nu respectă regulile standard ale algebrei. Deși mulți programatori sunt conștienți de acest lucru, aceștia din reflex aplică regulile algebrei clasice atunci când efectuează calcule cu numere reale. Pentru a preveni acest lucru aritmetica numerelor reale trebuie înțeleasă pentru a fi utilizată corect.

Normele algebrei clasice se aplică numai aritmeticii cu precizie infinită. Dacă, de exemplu, *x* și y sunt numere întregi, expresia *z=x+y* nu poate fi întotdeauna evaluată corect. Pe orice computer modern, această expresie (care respectă regulile algebrei clasice), este corectă atâta timp cât nu apare o depășire. Cu alte cuvinte *z=x+y* poate fi evaluat numai și numai pentru anumite valori care se încadrează în domeniul de reprezentare a numărului *z*.

Numerele întregi nu respectă regulile algebrei clasice, deoarece sistemele de calcul reprezintă numerele întregi pe un număr finit de biți. Valorile în virgulă fixă și cele în virgulă mobilă suferă de aceeași problemă, dar mai rău. La urma urmei, numerele întregi sunt un subset al numerelor reale. Prin urmare, valorile reale trebuie să reprezinte același set infinit de numere întregi. Există un număr infinit de valori între oricare două valori reale, deci această problemă este infinit mai gravă. Prin urmare, pe lângă faptul că numerele reale sunt pot fi reprezentate, ca și numerele întregi, doar între un interval maxim și minim, nu pot nici reprezenta toate valorile din acest interval.

1.2.3.1. Reprezentarea numerelor în virgula fixă

În reprezentarea în virgulă fixă, se consideră un număr finit de cifre semnificative, atât pentru partea întreagă a numărului, cât și pentru cea fracționară [2]. Plecând de la ecuația (1) un număr `x` în virgulă fixă, în baza `b` și care are `n` cifre pentru partea întreagă și *m* cifre pentru partea fracționară se reprezintă după expresia:

$$x = i_{n-1}*b^{n-1}+\ldots+i_2*b^2+i_1*b^1+i_0*b^0+ \\ +f_1*b^{-1}+f_2*b^{-3}+\ldots+f_{m-1}*b^{-(m-1)} \quad , \tag{3}$$

unde $i_{n-1}...i_0$ reprezintă cifrele părții întregi, iar cifrele `f1...fm` reprezintă partea fracționară a numărului `x`. Scrierea pozițională a acestui număr în baza *b* ar fi: $i_{n-1}i_{n-2}...i_0.f_1f_2...f_m$ (în tehnica de calcul virgula se notează cu caracterul punct '.').

Într-o reprezentare internă concretă baza de numerație `b` și numărul de cifre ale numărului (*n* și *m*) sunt alese convenabil pentru a satisface cerințele unei probleme date. De exemplu, dacă numerele vor fi reprezentate în baza 10 (`b=10`) și presupunând că o precizie de `0.00001` este suficientă, se vor păstra doar cinci cifre după virgulă (`m=5`). Similar dacă se presupune că numerele care trebuie reprezentate sunt pozitive și nu depășesc valoarea `100000000`, se vor aloca opt cifre pentru partea întreagă (`n=8`). Astfel numerele se vor reprezenta numere zecimale domeniul `0.00001`...`99999999.99999`.

Punctul zecimal (virgula) nu se salvează deoarece poziția acestuia este cunoscută (fixă) – de aici și numele acestui tip de reprezentare: virgulă fixă.

Operațiile aritmetice de înmulțire și împărțire produc valori aproximative. Înmulțind `1.12345` cu `2.57800` și obține valoarea teoretică `2.8962541` care nu poate fi reprezentată în

sistemul cu 5 cifre după virgulă. Valoarea rezultată poate fi stocată doar ca valoare aproximativă obținută prin rotunjire la `2.89625`. Operațiile de adunare și scădere sunt precise (rezultatul obținut este exact) cu condiția ca acesta să nu depășească limitele domeniului de reprezentare.

Reprezentarea în virgulă fixă se folosește în unele sisteme industriale (sisteme de poziționare) și în domeniul financiar contabil. Aplicațiile financiar contabile folosesc sistemul de reprezentare în virgula fixă datorită faptului că toate puterile lui 10 (pozitive și negative) care fac parte din domeniul de reprezentare sunt reprezentate exact.

1.2.3.2. Reprezentarea numerelor în virgulă mobilă

Reprezentarea în virgulă mobilă se utilizează cu precădere în domeniile științifice și tehnice, cu alte cuvinte în toate domeniile în afară de cel economico-financiar.

Pentru a reprezenta numere reale, cele mai multe formate de virgulă mobilă folosesc notația științifică (a se vedea Figura 1.1) și folosesc un număr de biți pentru a reprezenta o *mantisă* și un număr mai mic de biți pentru a reprezenta un *exponent*. Reprezentarea în virgulă mobilă se numește normalizată dacă se impune condiția ca cifra cea mai semnificativă (cea de dinainte de virgulă) să fie nenulă. Reprezentarea normalizată face imposibilă reprezentarea lui zero, dar oferă următoarele avantaje:

- reprezentarea fiecărui număr este unică
- nu de pierd biți pentru reprezentarea primelor zerouri de după virgulă
- în sistemele binare prima cifră nu necesită spațiu de stocare (deoarece este întotdeauna `1`).

Figura 1.1. Formatul pentru notația științifică.

Pentru a efectua operația de adunare și scădere a două numere în virgulă mobilă, trebuie ajustate cele două valori astfel încât exponenții lor să fie identici (valorile se de-normalizează). Din păcate, rezultatul nu se încadrează întotdeauna în cifre semnificative ale formatului, astfel încât rezultatul teoretic trebuie rotunjit sau trunchiat (în general, rotunjirea produce rezultatul cel mai precis). O altă problemă cu adunarea și scăderea este aceea se poate obține un rezultat în falsă precizie[1]. Pentru a evita acest lucru, unele (co)procesoare matematice sau pachete software pentru virgulă mobilă pot insera cifre aleatoare (sau biți) în cele mai nesemnificative poziții ale mantisei.

Înmulțirea și împărțirea nu suferă de aceleași probleme ca adunarea și scăderea, deoarece exponenții nu trebuiesc ajustați înaintea operației operație. Înmulțirea și împărțirea se realizează prin adunarea respectiv scăderea exponenților și înmulțirea respectiv împărțirea mantiselor. În sine, înmulțirea și împărțirea produc rezultate mulțumitoare. Cu toate acestea, ei tind să înmulțească orice eroare care exista deja într-o valoare. Înmulțirea și împărțirea nu sunt totuși lipsite de propriile lor probleme. Atunci se înmulțesc două numere foarte mari sau foarte mici, este foarte posibil să apară depășiri (*overflow* sau *underflow*). Aceeași situație poate apărea la împărțirea unui număr mic la un număr mare sau la un număr mare la un număr mic.

Compararea numerelor în virgulă mobilă este foarte ambiguă. Având în vedere inexactitățile prezente în orice calcul, comparația a două valori în virgulă mobilă pentru egalitate este de evitat. Într-un format binar în virgulă mobilă, calcule care produc același rezultat (matematic) pot diferi în biții cei mai nesemnificativi. Un test de egalitate reușește dacă și numai

1 Precizia falsă apare atunci când datele numerice sunt prezentate într-o manieră care implică o precizie mai bună decât este justificată, de exemplu când un număr real are mai multe zerouri după ultima cifră diferită de 0 (100.000%).

dacă toți biții (sau cifrele) din cei doi operanzi sunt exact aceiași. Întrucât acest lucru nu este neapărat valabil după două calcule diferite în virgulă nobilă care ar trebui (teoretic) să producă același rezultat, este posibil ca testul direct pentru egalitate să nu funcționeze. Modalitatea standard de a testa egalitatea între numerele în virgula mobilă este aceea de a determina care este eroarea (sau toleranța) permisă într-o comparație și să se verifice dacă o valoare se află în acest interval de eroare al celeilalte (sau dacă diferența dintre cele două numere în valoare absolută este mai mică decât eroarea).

1.2.3.3. Formatele IEEE pentru virgulă mobilă

Când proiectanții de la Intel și-au propus să introducă un co-procesor în virgulă mobilă pentru noul lor microprocesor 8086, aceștia au fost suficient de deștepți pentru a realiza că inginerii electricieni și fizicienii care proiectează cipuri nu sunt, poate, cei mai în măsură pentru a face o analiză numerică necesară pentru a alege cea mai bună reprezentare binară posibilă pentru un format în virgulă mobilă. Astfel, Intel, l-a angajat pe cel mai bun analist numeric pe care l-au putut găsi pentru a proiecta un format în virgulă mobilă pentru FPU[1] 8087. Acesta a angajat apoi alți doi experți în domeniu, iar cei trei (Kahn, Coonan și Stone) au conceput formatul în virgulă mobilă promovat de Intel. Au făcut o treabă atât de bună proiectând standardul KCS Floating Point Standard, încât organizația IEEE a adoptat acest format pentru formatul IEEE în virgulă mobilă[2].

Pentru a gestiona o gamă largă de cerințe de performanță și precizie, Intel a introdus de fapt trei formate în virgulă mobilă: precizie simplă, precizie dublă și precizie extinsă. Formate de precizie simplă și dublă corespund tipurilor C `float` și `double`. Intel intenționa să utilizeze o precizie extinsă pentru secvențe lungi de calcule. Precizia extinsă conține 16 biți suplimentari pe care calculele îi pot utiliza pentru valori intermediare înainte de rotunjirea rezultatului final la o valoare în dublă precizie.

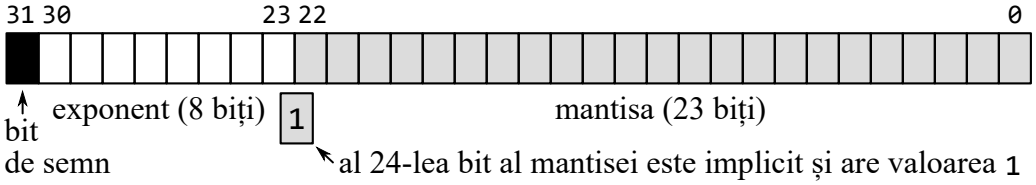

Figura 1.2. Formatul cu precizie simplă pe 32 biți.

Formatul în simplă precizie utilizează o mantisă de 24 biți și un exponent de 8 biți în reprezentare Excess-127. Mantisa reprezintă, de obicei, o valoare cuprinsă între `1.0` și până la `2.0`. Cel mai semnificativ bit al mantisei se presupune întotdeauna ca fiind `1` și reprezintă valoarea de la stânga punctului binar; restul de 23 biți ai mantisei apar în dreapta punctului binar. Prin urmare, mantisa reprezintă valoarea (unde m_{22} ... m_0 reprezintă biții mantisei):

$$1.m_{22}m_{21}m_{20}m_{19}m_{18}m_{17}m_{16}m_{15}m_{14}m_{13}m_{12}m_{11}m_{10}m_9m_8m_7m_6m_5m_4m_3m_2m_1m_0$$

Deși există un număr infinit de valori în intervalul `[1.0, 2.0)`, se pot reprezenta doar opt milioane dintre acestea, deoarece mantisa are 23 biți (cel de-al 24-lea bit este întotdeauna unul). Acesta este motivul inexactității în aritmetica în virgulă mobilă – cei 23 biți limitează precizie în calculele care implică valori cu precizie simplă.

Mantisa folosește formatul complement față de 1, și nu complement față de 2. Numerele

1 FPU – din engl. Floating Point Unit (unitate în virgulă mobilă)

2 Sunt unele modificări minore în modul în care sunt fost gestionate anumite operații, dar reprezentarea biților a rămas esențial neschimbată.

în complement față de 1 au proprietatea că există două reprezentări pentru zero (un `+0` și un `-0`).

Pentru a reprezenta valori în afara intervalului `[1.0, 2.0)`, intră în joc exponentul formatului în virgulă mobilă. Formatul în virgulă mobilă multiplică valoarea mantisei cu `2` la puterea specificată de exponent. Exponentul este de 8 biți și este stocat într-un format Excess-`127`. În formatul Excess-`127`, exponentul 2^0 este reprezentat de valoarea `127` (`7Fh`). Prin urmare, pentru a converti un exponent în format Excess-`127`, pur și simplu se adaugă `127` la valoarea exponentului. Utilizarea formatului Excess-`127` facilitează compararea valorilor în virgulă mobilă. Formatul în virgulă mobilă cu precizie simplă (32 biți) ia forma prezentată în Figura 1.2.

Cu o mantisă de 24 biți, se pot obține aproximativ 6½ cifre de precizie (o jumătate de cifră de precizie înseamnă că primele șase cifre pot fi toate în intervalul `0...9`, dar a șaptea cifră poate fi doar în intervalul `0...x` unde `x<9` și este în general aproape de cinci). Cu un exponent pe 8 biți în format Excess-`127`, intervalul dinamic al numerelor în virgulă mobilă precizie simplă este de aproximativ $2^{\pm 128}$ sau aproximativ $10^{\pm 38}$.

Deși numerele în virgulă mobilă cu precizie simplă sunt perfect adecvate pentru multe aplicații, intervalul dinamic este oarecum mic pentru multe aplicații științifice, iar precizia foarte limitată nu este potrivită pentru multe aplicații financiare, științifice și alte aplicații. Mai mult, în secvențele lungi de calcule, precizia limitată a formatului cu precizie simplă poate introduce (și propaga) erori semnificative.

Formatul cu precizie dublă ajută la depășirea limitărilor impuse de formatul cu precizie simplă. Folosind de două ori spațiul de stocare, formatul precizie dublă de are un exponent de 11 biți în format Excess-`1023` și o mantisă de 53 biți (cu tot cu bitul cel mai semnificativ implicit cu valoarea `1`), plus un bit de semn. Acest format oferă o gamă dinamică de aproximativ $10^{\pm 308}$ și 14½ cifre de precizie, suficiente pentru majoritatea aplicațiilor. Formatul în virgulă mobilă cu precizie dublă (64 biți) ia forma prezentată în Figura 1.3.

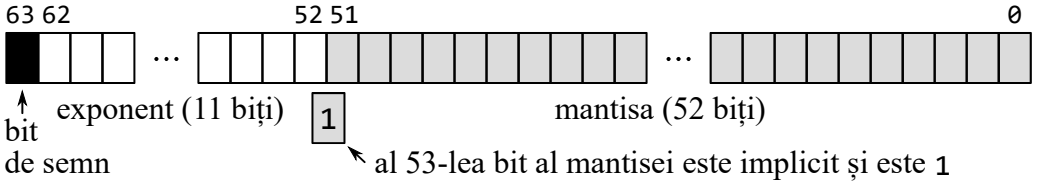

Figura 1.3. Formatul cu precizie dublă pe 64 biți.

Pentru a îmbunătăți precizia pentru secvențele lungi de calcule care implică numere în virgulă mobilă, Intel a conceput formatul cu precizie extinsă. Formatul cu precizie extinsă folosește 80 biți. Doisprezece (12) dintre cei 16 biți suplimentari sunt adăugați mantisei, iar patru (4) sunt adăugați exponentului. Spre deosebire de formatul cu precizie simplă sau dublă, formatul cu precizie extinsă nu are cel mai semnificativ bit al mantisei implicit (bit care era întotdeauna `1`). Prin urmare, formatul cu precizie extinsă oferă o mantisă pe 64 biți, un exponent de 15 biți în Excess-`16383` și un bit de semn. Formatul în virgulă mobilă cu precizie extinsă (80 biți) ia forma prezentată în Figura 1.4.

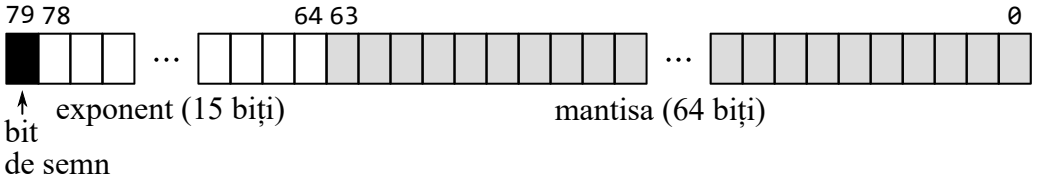

Figura 1.4. Formatul cu precizie extinsă pe 80 biți.

În co-procesorul matematic 80x87, toate calculele sunt efectuate folosind formatul cu precizie extinsă. Ori de câte ori este încărcată o valoare cu precizie simplă sau dublă, co-procesorul o transformă automat într-o valoare cu precizie extinsă. De asemenea, atunci când se salvează în memorie o valoare în simplă sau dublă de precizie, acesta rotunjește automat valoarea la dimensiunea corespunzătoare înainte de a efectua transferul. Lucrând întotdeauna cu formatul de precizie extins, Intel garantează că sunt prezenți un număr mai mare de biți suplimentari pentru a asigura exactitatea calculelor. Prin efectuarea tuturor calculelor folosind 80 biți, Intel asigură (dar nu garantează) că se va obține o precizie completă pe 32 sau 64 biți în efectuarea calculelor. Întrucât co-procesorul matematic 80x87 nu furnizează un număr mare de biți suplimentari în calculele pe 80 biți, inevitabil o eroare se va strecura în cei mai nesemnificativi biți ai unui calcul cu precizie extinsă. Cu toate acestea, dacă calculul este corect pe 64 biți, calculul pe 80 biți va oferi întotdeauna cel puțin 64 biți exacți.

Pentru a menține precizia maximă în timpul calculului, majoritatea calculelor folosesc valori normalizate. O valoare normală virgulă mobilă este cea care are cel mai semnificativ bit al mantisei egal cu `1`. Aproape orice valoare ne-normalizată poate fi normalizată prin deplasarea biților mantisei spre stânga și decrementarea exponentului cu `1` până când un bit de `1` apare cel mai semnificativ bit al mantisei.

Păstrarea numerelor în în virgulă mobilă în forma lor normală are avantajul că oferă numărul maxim de biți de precizie pentru efectuarea calculelor. Dacă cei mai semnificativi biți ai mantisei sunt zero, mantisa va avea cu atât mai puțini biți disponibili pentru preciza calculului. Prin urmare, un calcul în virgulă mobilă va fi mai precis dacă implică doar valori normalizate.

Există două cazuri importante în care un număr de în virgulă mobilă nu poate fi normalizat. Valoarea `0.0` este un caz special. Evident, aceasta nu poate fi normalizată, deoarece reprezentarea în virgulă mobilă nu are nici un bit de `1` în mantisă. În standardul IEEE 754, valorile zero sunt reprezentate de exponentul și mantisa cu toți biții pe `0`. Zero negativ (`-0.0`) are bitul de semn setat pe `1`, iar zero pozitiv va avea bitul de semn setat pe `0`.

Al doilea caz apare atunci când câțiva din cei mai semnificativi biți ai mantisă care sunt zero, dar exponentul este de asemenea zero (cel mai negativ exponent posibil). Astfel exponentul nu mai poate fi decrementat pentru a normaliza mantisa. Standardul IEEE 754 permite anumite valori speciale de-normalizate pentru a reprezenta aceste valori foarte mici. Deși utilizarea valorilor de-normalizate permite calculelor IEEE în virgulă mobilă să producă rezultate mai bune decât în cazul în care s-ar produce depășire (*underflow*), trebuie reținut faptul că valorile de-normalizate oferă mai puțini biți de precizie și sunt în mod inerent mai puțin exacte.

Deoarece co-procesorul matematic 80x87 convertesc întotdeauna valorile de precizie simplă și dublă în precizie extinsă, aritmetica cu precizie extinsă este de fapt mai rapidă decât cea cu precizie singură sau dublă. Prin urmare, beneficiile preconizate ale utilizării formatelor mai mici nu sunt prezente pe aceste cipuri. Cu toate acestea, începând cu procesorul Pentium, Intel a reproiectat unitatea în virgulă mobilă încorporată pentru a concura mai bine cu cipurile RISC[1]. Majoritatea cipurilor RISC acceptă un format nativ de precizie dublă pe 64 biți, care este mai rapid decât formatul de precizie extins al Intel. Prin urmare, Intel a furnizat operațiuni native pe 64 biți pe Pentium pentru a concura mai bine cu cipurile RISC. Prin urmare, formatul dublu de precizie este cel mai rapid pe cipurile Pentium și ulterior.

1.3. Algebra Boole

Algebra numerică este cunoscută din școala generală. Astfel se cunoaște ce este acela un

1 RISC – din engl. Reduced Instruction Set Computer, sunt microprocesoare cu set redus de instrucțiuni, este o arhitectură cu un set de comenzi simple și rapide, în care viteza crește datorită simplității instrucțiunilor.

set de valori sau domeniu (de exemplu: $\mathbb{Z}$ mulțimea numerelor întregi, $\mathbb{R}$ mulțimea numerelor reale), o listă de operatori (de exemplu „+" operatorul adunare, „−, scădere sau „·" înmulțire) sau o listă de axiome care dictează cum funcționează acești operatori (poate nu ne amintim exact denumirea acestor axiome dar știm cum funcționează, de exemplu: fie $x,y,z\in\mathbb{Z}$, atunci $x+(y+z)=(x+y)$.

La începutul anilor 1940, matematicianul George Boole a combinat logica propozițională și teoria mulțimilor pentru a forma un sistem pe care noi astăzi îl numim algebra booleană. Exprimat altfel Boole a observat că a opera cu o expresie x este echivalent cu a lucra cu un număr y.

Astfel algebra booleană are:

- un set de valori: {adevărat, fals};
- o listă de operatori: $\bar{}$, $\vee$ și $\wedge$;
- o listă de axiome care dictează cum funcționează operatorii $\bar{}$, $\vee$ și $\wedge$.

Ironic este faptul că la momentul respectiv ideea lui Boole a fost văzută ca oarecum obscură și chiar însuși Boole nu a văzut logica ca un concept matematic. Totuși, acest pas spre identificarea punctelor comune între aceste două concepte și unificarea lor a fost o unealtă puternică în științele matematice, oferind o viziune generală asupra a ceea ce la prima vedere pare a fi diferit. Abia în 1937, Claude Shannon, la momentul respectiv student atât la inginerie electrică cât și la matematică, a văzut potențialul utilizării algebrei booleene în reprezentarea și manipularea informației digitale.

Definiție: Un *operator binar* $\odot$ pe un set de valori $\mathbb{A}$ se definește ca o funcție $\odot : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}$, iar un operator unar $\oslash$ ca o funcție: $\oslash : \mathbb{A} \rightarrow \mathbb{A}$.

Definiție: Se definește setul de valori $\mathbb{B} = \{0, 1\}$ în care există doi operatori binari $\vee$, $\wedge$ și un operator unar $\bar{}$. Membrii mulțimii $\mathbb{B}$ sunt elemente neutre pentru operatorii $\vee$ respectiv $\wedge$. Operatorii $\bar{}$, $\vee$ și $\wedge$ numiți NU (NOT), ȘI (AND) respectiv SAU (OR) sunt guvernați de următoarele axiome:

echivalența	$x \leftrightarrow y$	$\equiv (x \rightarrow y) \wedge (y \rightarrow x)$
implicația	$x \rightarrow y$	$\equiv \bar{x} \vee y$
involuția	$\bar{\bar{x}}$	$\equiv x$
idempotența	$x \wedge x$	$\equiv x$
comutativitatea	$x \wedge y$	$\equiv y \wedge x$
asociativitatea	$(x \wedge y) \wedge z$	$\equiv x \wedge (y \wedge z)$
distributivitatea	$x \wedge (y \vee z)$	$\equiv (x \wedge y) \vee (x \wedge z)$
de Morgan	$\overline{x \wedge y}$	$\equiv \bar{x} \vee \bar{y}$
identitatea	$x \wedge 1$	$\equiv x$
elementul neutru	$x \wedge 0$	$\equiv 0$
complementaritatea	$x \wedge \bar{x}$	$\equiv 0$
absorbția	$x \wedge (x \vee y)$	$\equiv x$
idempotența	$x \vee x$	$\equiv x$
comutativitatea	$x \vee y$	$\equiv y \vee x$
asociativitatea	$(x \vee y) \vee z$	$\equiv x \vee (y \vee z)$
distributivitatea	$x \vee (y \wedge z)$	$\equiv (x \vee y) \wedge (x \vee z)$
de Morgan	$\overline{x \vee y}$	$\equiv \bar{x} \wedge \bar{y}$
identitatea	$x \vee 0$	$\equiv x$
elementul neutru	$x \vee 1$	$\equiv 1$
complementaritatea	$x \vee \bar{x}$	$\equiv 1$
absorbția	$x \vee (x \wedge y)$	$\equiv x$

Se observă cum această descriere a algebrei booleene unește conceptele de logică propozițională și teoria mulțimilor: 0 și „fals” și $\emptyset$ (mulțimea vidă) sunt echivalente, precum 1 și „adevărat” și mulțimea universală.

Se observă că operatorii $\vee$ și $\wedge$ din algebra booleană se comportă într-un mod similar cu operatorii $\cdot$ și + din algebra numerică. Astfel, operatorii $\vee$ și $\wedge$ sunt deseori numiți (și chiar notați) ca „produs” și „sumă”.

1.3.1. Tabele de adevăr

O tabelă de adevăr indică valoarea unei funcții pentru toate combinațiile posibile ale valorilor variabilelor de intrare ale funcției. Câte o coloană a tabelului de adevăr va corespunde fiecărei variabile și o coloana pentru valoarea funcției. Deoarece fiecare variabilă poate lua două valori (1 sau 0), numărul de combinații de valori se multiplică în funcție de numărul variabilelor de intrare (pentru *n* variabile de intrare binare vom avea 2^n combinații). De exemplu, dacă există doua variabile, vor avea $2^2 = 4$ combinații ale valorilor variabilelor de intrare și corespunzător, 4 rânduri în tabela de adevăr. Dacă expresia din membrul drept a funcției este complexă, tabela de adevăr poate fi construită în mai mulți pași. Exemplul următor ilustrează construcția unei tabele de adevăr.

Exemplu: Fie tabela de adevăr a funcției $Z=A\cdot\overline{B}+\overline{A}\cdot C+\overline{A}\cdot\overline{B}\cdot C$.

Funcția are 3 variabile de intrare: A, B și C, deci vom avea 2^3 combinații ale valorilor variabilelor de intrare A, B și C. Cele 8 combinații sunt afișate în partea stângă a tabelei de adevăr din Tabelul 1.7. Aceste combinații sunt în ordine crescătoare în sistemul de numerație binar, începând de la $000_{(2)}$ (sau $0_{(10)}$ în sistem zecimal) până la $111_{(2)}$ ($7_{(10)}$). În general dacă avem n variabile de intrare vom aveam 2^n combinații cu valori între 0 și 2^{n-1}.

Tabelul 1.7. Tabelul de adevăr al funcției $Z=A\cdot B+A\cdot C+A\cdot B\cdot C$

A	B	C	$\overline{A}$	$\overline{B}$	$A\cdot\overline{B}$	$\overline{A}\cdot C$	$\overline{A}\cdot\overline{B}\cdot C$	Z
0	0	0	1	1	0	0	0	0
0	0	1	1	1	0	1	1	1
0	1	0	1	0	0	0	0	0
0	1	1	1	0	0	1	0	1
1	0	0	0	1	1	0	0	1
1	0	1	0	1	1	0	0	1
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0

Pentru evaluarea funcției Z pe baza valorilor variabilelor de intrare se va calcula mai întâi valorile pentru coloanele $\overline{A}$ și $\overline{B}$, apoi valorile pentru $A\cdot\overline{B}$, $\overline{A}\cdot C$ și $\overline{A}\cdot\overline{B}\cdot C$ folosind funcția logică ȘI (AND) pentru valorile din coloanele corespunzătoare variabilelor din expresie, iar în final valoarea pentru funcția Z este obținută prin aplicarea funcției logice SAU (OR) ultimelor 3 coloane ale tabelului. Evaluarea coloanei $\overline{A}\cdot\overline{B}\cdot C$ corespunde folosirii funcției ȘI (AND) coloanelor $\overline{A}$ și B, urmată de folosirea funcției ȘI (AND) la valorile coloanei C (sau poate fi utilizată coloana deja existentă $\overline{A}\cdot C$ asupra căreia se va efectua un ȘI logic cu coloana $\overline{B}$). Coloanele corespunzătoare expresiilor intermediare $A\cdot\overline{B}$, $\overline{A}\cdot C$ și $\overline{A}\cdot\overline{B}\cdot C$ nu sunt de obicei prezente în tabelele de adevăr.

1.3.2. Formele normale ale expresiilor booleene

Definiție: Fie o funcție booleană *f* cu *n* intrări, de exemplu $f(x_0, x_2, \ldots, x_{n-1})$. Când expresia

pentru funcția *f* este scrisă ca o sumă (funcția logică SAU - OR) de termeni fiecare compus din produsul (funcția logică ȘI - AND) variabilelor de intrare se spune ca această expresie este „forma normal disjunctivă" sau „suma produselor" SoP (Sum of Products) termenii acestei expresii se numesc mintermi. Când expresia pentru funcția *f* este scrisă ca un produs (funcția logică ȘI - AND) de termeni fiecare compus din suma (funcția logică SAU - OR) variabilelor de intrare, se spune ca această expresie este „forma normal conjunctivă" sau „produsul sumelor" PoS (Product of Sums) termenii acestei expresii se numesc maxtermi.

Aceasta este o definiție formală pentru un fapt foarte ușor de descris prin exemplu. Fie funcția *f* descrisă de tabelul de mai jos:

x	*y*	*f(x,y)*
0	0	0
0	1	1
1	0	1
1	1	0

Mintermii (valorile de 1) sunt pe linia a doua și a treia a tabelului de adevăr. Maxtermii (valorile de 0) sunt pe prima și ultima linie a tabelului. Pentru a obține o formă normal disjunctivă din tabela de adevăr, putem folosi următoarea procedură: expresia va fi sub forma unei sume de produse, pentru fiecare rând din tabelul de adevăr în care valoarea funcției este 1 vom obține un minterm (termen produs) pentru care se consideră variabilele individuale ca fiind nenegate dacă valoarea variabilei pe acel rând este 1 și negate (complementate) dacă valoarea variabilei pe acel rând este 0. În mod similar pentru a obține o formă normal conjunctivă din tabela de adevăr expresia va fi sub forma unui produs de sume, pentru fiecare rând din tabelul de adevăr în care valoarea funcției este 0 vom obține un maxterm (termen sumă) pentru care se consideră variabilele individuale ca fiind nenegate dacă valoarea variabilei pe acel rând este 0 și negate (complementate) dacă valoarea variabilei pe acel rând este 1. O expresie a funcției *f* în forma normal disjunctivă (SoP) și forma normal conjunctivă (PoS) este: $f_{SoP}(x,y)=(\bar{x}\cdot y)+(x\cdot\bar{y})$ respectiv $f_{PoS}(x,y)=(x+y)(\bar{x}+\bar{y})$.

Deși nu par deosebit de interesante la o prima vedere, aceste expresii pun la dispoziție o tehnică foarte utilă. Pe baza tabelului de adevăr al unei funcții arbitrară, se poate construi o expresie a funcției doar prin extragerea mintermilor sau maxtermilor și construirea formei normal disjunctive sau conjunctive.

Capitolul 2. Arhitectura setului de instrucțiuni

Programarea în limbaj de asamblare este dependentă de arhitectura sistemului de calcul pentru care se dezvoltă aplicația software.

Arhitectura setului de instrucțiuni (ISA[1]) reprezintă partea din arhitectura unui calculator care reprezintă interfața prin care acesta poate fi programat, și conține tipuri de date, instrucțiuni, regiștri, modurile de adresare, arhitectura memoriei, rutinele de tratare a întreruperilor și excepțiilor, și intrările și ieșirile standard. Arhitectura setului de instrucțiuni conține și o specificație a codurilor operațiilor, comenzile native implementate de o anume unitate de procesare [3]. *Setul de instrucțiuni* reprezintă o listă a tuturor instrucțiunilor pe care le poate executa un procesor. *Procesorul* (CPU[2]) este dispozitivul care execută instrucțiunile programului în cod mașină. *Instrucțiunea* este o comandă primitivă către procesorului (de exemple: mutarea datelor între regiștri, accesul la memorie, operații aritmetice de bază). *Codul mașină* reprezintă secvența de cod pe care procesorul o prelucrează direct direct. Fiecare instrucțiune este de obicei codată de mai mulți octeți (*bytes*).

În limbajul de asamblare programatorul are acces direct la regiștrii interni ai procesorului. Fiecare CPU dispune de un set fix de regiștri de uz general (8 în x86, 16 în x86-64 și de asemenea 16 în ARM). Un *registru procesor* este o cantitate mică de spațiu de stocare disponibilă în unitatea centrală de procesare, spațiu al cărui conținut poate fi accesat mai rapid decât datele aflate în alte resurse (de exemplu, în memoria principală).

Arhitectura setului de instrucțiuni pentru procesoarele x86 a avut întotdeauna instrucțiuni de lungime variabilă, astfel încât atunci când a venit era „64 biți”, extensiile x64 nu au avut un impact semnificativ asupra ISA. De fapt, ISA x86 conține încă o mulțime de instrucțiuni apărute la procesorul pe 16 biți 8086, și totuși se regăsesc și în procesoarele actuale.

De exemplu, ARM este un procesor RISC[3] proiectat cu instrucțiuni de lungime constantă, care au avut unele avantaje în trecut. La început, instrucțiunile ARM au fost codificate pe 4 octeți. Acest lucru este denumit acum "modul ARM". Ulterior s-a observat că nu este atât de eficient pe cât s-au așteptat inițial dezvoltatorii. De fapt, cele mai instrucțiuni uzuale ale procesorului în aplicațiile uzuale pot fi codificate folosind mai puțini biți. Prin urmare, ei au adăugat un alt ISA, numit Thumb, în care fiecare instrucțiune a fost codificată în doar 2 octeți. Acest lucru este denumit acum "modul Thumb". Cu toate acestea, nu *toate* instrucțiunile ARM pot fi codificate în doar 2 octeți, deci setul de instrucțiuni Thumb este puțin limitat. Merită menționat faptul că codul compilat pentru modul ARM și modul Thumb pot coexista într-un singur program. Creatorii ARM au crezut că Thumb ar putea fi extins, dând naștere la Thumb-2, care a apărut în ARMv7. Thumb-2 utilizează în continuare instrucțiuni de 2 octeți, dar are câteva instrucțiuni noi care au dimensiunea de 4 octeți. Ulterior a apărut ARM-ul pe 64 biți. Acest ISA are instrucțiuni de 4 octeți și nu are nevoie de nici un mod Thumb suplimentar. Cu toate acestea, cerințele pe 64 biți au afectat ISA, ceea ce a dus la existența a trei seturi de instrucțiuni ARM: modul ARM, modul Thumb (inclusiv Thumb-2) și ARM64. Aceste ISA intersectează parțial, dar nu se poate spune că acestea sunt ISA-uri diferite, mai degrabă decât variații ale aceleiași ISA. Există chiar multe alte ISA RISC cu instrucțiuni pe 32 biți cu lungime fixă, cum ar fi MIPS, PowerPC și Alpha AXP.

1 ISA – din engl. Instruction Set Architecture (Arhitectura setului de instrucțiuni).
2 CPU – din engl. Central Procesing Unit (Unitatea centrală de procesare).
3 RISC – din engl. Reduced Instruction Set Computing

2.1. Limbajul de asamblare. Noțiuni generale

De ani buni, s-a prezis dispariția limbajului de asamblare, susținând că lumea este în sfârșit gata să treacă la abordări mai puțin primitive în ceea ce privește programarea... și de ani buni, cele mai bune programe au fost scrise, măcar parțial, în asamblare. De ce asta? Pur și simplu pentru că deși limbajul de asamblare este greu de folosit, dar – utilizat corect – produce programe de performanță fără egal. Studierea limbajul de asamblare nu te va face cu siguranță un expert în programare – dar fără el nu îți vei atinge niciodată potențialul maxim de programator. De ce limbajul de asamblare este atât de important în această epocă a compilatoarelor optimizate și generatoarelor de programe? Limbajul de asamblare este fundamental diferit de toate celelalte limbaje de programare, acesta permite utilizarea fiecărei „fărâme" de resursă a calculatorului pentru a atinge vârful de performanță.

Limbajele de nivel înalt sunt cu siguranță mai ușor de utilizat, iar în prezent, majoritatea permit accesul la resursele computer-ului fără a fi nevoie să apelăm la asamblare Dacă, pe de altă parte, se dorește ca programul să ofere programelor interfețe rapide și timpi de răspuns cât mai mici, limbajul de asamblare se dovedește a fi aproape magic, pentru că nici un alt limbaj nu se apropie de asamblare pentru o viteză pură.

Desigur, nimeni nu testează limitele calculatorului-ului cu primul lor program de asamblare; acest lucru necesită timp și practică. În timp ce mulți programatori de PC știu câte ceva despre asamblare, puțini sunt experți. Programatorul tipic a scris cod de asamblare preluat dintr-un articol sau de pe internet, a citit o carte despre programarea asamblorului și a scris poate câteva programe de asamblare ale sale – dar încă nu simte că a stăpânit limbajul.

Codul în asamblare este predispus la erori, greu de depanat, dificil de realizat într-un mod clar structurat, greu de citit și dificil de întreținut. Înainte de a începe programarea în asamblare, trebuie determinat ce parte a programului poate fi realizat în asamblare și ce metodă de programare trebuie utilizată. Dacă strategia de dezvoltare nu este clară, se va pierde timp optimizând părți greșite ale programului, făcând lucruri în asamblare care ar fi putut fi realizate în C++, încercând să se optimizeze lucruri care nu mai pot fi optimizate, scriind secvențe de cod care sunt greu de întreținut și care sunt pline de erori și dificil de depanat.

2.1.1. Avantajele asamblării

Programarea în asamblare nu se folosește în prezent fel de mult ca în trecut. Cu toate acestea, există încă motive pentru învățarea și utilizarea codului de asamblare. Principalele motive sunt:

1. Motive educaționale. Este importantă cunoașterea medului de funcționare ale microprocesoarelor și compilatoarelor la nivel de instrucțiuni pentru a putea prezice ce tehnici de codare sunt cele mai eficiente, pentru a înțelege modul în care funcționează diferite construcțiile în limbajele la nivel înalt și pentru a urmări erorile greu de găsit.
2. Depanare și verificare. Analiza codului de asamblare generat de compilator sau fereastra de dezasamblare într-un depanator este utilă pentru a găsi erori și pentru a verifica cât de bine un compilator optimizează o anumită secvență de cod.
3. Realizarea de compilatoare. Înțelegerea tehnicilor de codare a asamblării este necesară pentru realizarea compilatoarelor, depanatoarelor și a altor instrumente de dezvoltare.
4. Sisteme încorporate. Sistemele mici încorporate au mai puține resurse decât computerele. Programarea în asamblare poate fi necesară pentru optimizarea codului pentru viteză sau dimensiune în sisteme încorporate mici.

5. Drivere hardware și cod de sistem. Accesarea hardware-ului, a regiștrilor de control ai sistemului, etc. poate fi uneori dificilă sau imposibilă cu un cod de nivel înalt.
6. Accesarea instrucțiunilor care nu sunt accesibile din limbaj la nivel înalt. Anumite instrucțiuni de asamblare nu au un limbaj echivalent la nivel înalt.
7. Cod auto-modificabil. În general, codul auto-modificabil nu este profitabil, deoarece interferă cu funcționarea eficientă a cache-ului. Cu toate acestea, poate fi avantajos, de exemplu, includerea unui mic compilator în programe de matematică, unde o funcție definită de utilizator trebuie calculată de mai multe ori.
8. Optimizarea codului pentru dimensiuni. Spațiul de stocare și memoria sunt atât de ieftine în zilele noastre, încât nu merită efortul de a folosi limbajul de asamblare pentru a reduce dimensiunea codului. Cu toate acestea, dimensiunea memoriei cache este încă o resursă atât de critică încât poate fi utilă în unele cazuri pentru a optimiza o bucată critică de cod pentru dimensiune, pentru a o face să se încadreze în memoria cache de program.
9. Optimizarea codului pentru viteză. Compilatoarele moderne C++ optimizează în general codul destul de bine în majoritatea cazurilor. Dar există încă cazuri în care compilatoarele au performanțe slabe și în care se pot realiza creșteri dramatice ale vitezei printr-o programare atentă în asamblare.
10. Biblioteci de funcții. Beneficiul total al optimizării codului este mai mare în bibliotecile de funcții care sunt utilizate de mulți programatori.
11. Crearea bibliotecilor de funcții compatibile cu mai multe compilatoare și sisteme de operare. Este posibilă implementarea funcților de bibliotecă cu mai multe intrări compatibile cu diferite compilatoare și diferite sisteme de operare. Aceasta necesită programare în asamblare.

2.1.2. Dezavantajele asamblării

Există atât de multe dezavantaje și probleme implicate în programarea în asamblare, încât este indicat să ia în considerare alternativele înainte de a decide utilizarea codul de asamblare pentru o anumită sarcină. Cele mai importante motive pentru care nu folosiți programarea asamblării sunt:

1. Timp de dezvoltare. Scrierea codului în limbajul de asamblare durează mult mai mult timp decât într-un limbaj de nivel înalt.
2. Fiabilitate și securitate. Este ușor să se genereze erori în scrierea codul în asamblare. Asamblorul nu verifică dacă sunt respectate convențiile de apelare și dacă sunt salvați regiștrii utilizați. Nimeni nu vă verifică dacă numărul de instrucțiuni `PUSH` și `POP` este același în toate ramurile și căile posibile ale programului. Există atât de multe posibilități de erori ascunse în codul de asamblare, încât afectează fiabilitatea și securitatea proiectului, cu excepția cazului în care aveți o abordare foarte sistematică urmată de o testare și verificare amănunțită.
3. Depanare și verificare. Codul de asamblare este mai dificil de depanat și de verificat, deoarece există mai multe posibilități de a genera erori decât în codul de nivel înalt.
4. Mentenanță. Codul de asamblare este mai dificil de modificat și de întreținut, deoarece limbajul permite scrierea de cod nestructurat și tot felul de trucuri, dificile

pentru alți programatori. Este necesară o documentare detaliată și un stil de programare consecvent.

5. Codul de sistem poate utiliza funcții intrinseci în loc de asamblare. Cele mai bune compilatoare C++ moderne au funcții intrinseci pentru accesarea regiștrilor de control ai sistemului și a altor instrucțiuni ale sistemului. Codul de asamblare nu mai este necesar pentru driverele de dispozitiv și alte coduri de sistem atunci când sunt disponibile funcții intrinseci.
6. Codul aplicației poate utiliza funcții intrinseci sau clase de vectori în loc de asamblare. Cele mai bune compilatoare C++ moderne au funcții intrinseci pentru operațiile vectoriale și alte instrucțiuni speciale care au necesitat anterior programarea în asamblare. Nu mai este necesară utilizarea codului în asamblare de modă veche pentru a beneficia de instrucțiunile SIMD (Single-Instruction-Multiple-Data).
7. Portabilitatea. Codul de asamblare este foarte specific platformei. Portarea la o platformă diferită este dificilă. Codul care folosește funcții intrinseci în loc de asamblare sunt portabile la toate platformele x86 și x86-64.
8. Compilatoarele s-au îmbunătățit foarte mult în ultimii ani. Majoritatea compilatoarelor moderne sunt acum mai bune decât programatorul mediu de asamblare în multe situații.
9. Codul compilat poate fi mai rapid decât codul de asamblare, deoarece compilatoarelor pot realiza optimizarea inter-procedurală și optimizarea întregului program. Programatorul de asamblare, de obicei, trebuie să efectueze funcții bine definite cu o interfață de apel bine definită, care respectă toate convențiile de apelare pentru ca secvența de cod să poată fi testată și verificată. Acest lucru împiedică multe dintre metodele de optimizare pe care le utilizează compilatoarele, cum ar fi *inlining*-ul de funcții, alocarea regiștrilor, propagarea constantelor, eliminarea comună a sub-expresiilor între funcții, programarea între funcții, etc. Aceste avantaje pot fi obținute folosind codul C++ cu funcții intrinseci în loc de cod de asamblare.

Programele în ziua de azi sunt complexe. Nu este productiv programarea întregul program în asamblare. Aceasta este o pierdere de timp. Codul în asamblare trebuie utilizat numai acolo unde viteza este critică și unde se poate obține o îmbunătățire semnificativă a vitezei. Cea mai mare parte a programului poate fi realizată în C sau C++. Acestea sunt limbajele de programare care se combină cel mai ușor cu codul de asamblare.

2.2. Organizarea calculatoarelor PC

Pentru a scrie chiar și un modest program de limbaj de asamblare x86 necesită o familiaritate considerabilă cu familia de procesoare x86. Pentru a scrie programe bune de limbaj de asamblare necesită o cunoaștere temeinică a *hardware*-ului de bază. Din păcate, *hardware*-ul de bază nu este consecvent. Este posibil ca tehnicile care sunt cruciale pentru programele 486 să nu fie utile pe sistemele Pentium. De asemenea, tehnicile de programare care asigură îmbunătățiri de performanță pe un cip s-ar putea să nu ajute deloc pe un altul. Din fericire, unele tehnici de programare funcționează bine indiferent ce microprocesor utilizați.

Proiectarea operațională de bază a unui sistem informatic se numește *arhitectura sistemului*. John Von Neumann, un pionier în proiectarea computerelor, este considerat părintele arhitecturii majorității computerelor folosite astăzi. De exemplu, familia de procesoare x86 folosește *arhitectura Von Neumann*. Un sistem tipic Von Neumann are trei componente majore:

unitatea centrală de procesare (sau CPU), memoria și dispozitive de intrare/ieșire (sau I/O[1]). Modul în care un proiectant de sistem combină aceste componente afectează performanța sistemului (a se vedea Figura 2.1).

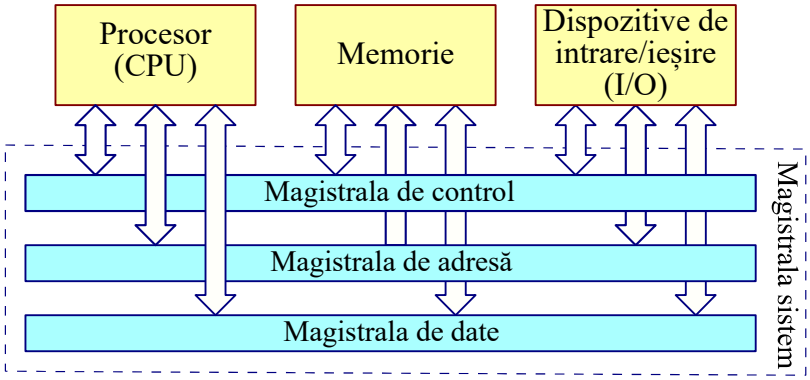

Figura 2.1. Structura arhitecturii Von Newman.

În mașinile Von Neumann, precum familia x86, procesorul este locul unde se desfășoară toată acțiunea. Toate calculele se realizează în interiorul procesorului. Instrucțiunile și datele se află în memorie până când sunt solicitate de CPU. Pentru CPU, majoritatea dispozitivelor I/O arată ca o memorie, deoarece CPU poate stoca date pe un dispozitiv de ieșire și poate citi date de pe un dispozitiv de intrare. Diferența majoră dintre locațiile de memorie și I/O este faptul că locațiile I/O sunt, în general, asociate cu dispozitive externe (periferice).

2.2.1. Magistrala sistemului (System bus)

Magistrala de sistem conectează diferitele componente ale unei mașini Von Neumann. Familia x86 are trei magistrale majore: magistrala de adrese, magistrala de date și magistrala de control. O *magistrală* este o colecție de fire pe care semnalele electrice trec între componentele din sistem. Aceste magistrale variază de la procesor la procesor. Cu toate acestea, fiecare magistrală poartă informații comparabile despre toate procesoarele; de exemplu, magistrala de date poate avea o implementare diferită pe 80386 decât pe 8088, dar ambele vehiculează date între procesor, dispozitivele I/O și memorie.

2.2.1.1. Magistrala de date

Procesoarele x86 folosesc magistrala de date pentru a vehicula date între diversele componente ale unui sistem de calcul. Mărimea acestei magistrale variază mult în familia x86. Într-adevăr, acestă magistrală definește „dimensiunea” procesorului. Pe sistemele tipice x86, magistrala de date conține 8, 16, 32 sau 64 de linii. Microprocesoarele 8088 și 80188 au 8 biți (opt linii de date); procesoarele 8086, 80186, 80286 și 80386SX au 16 biți; procesoarele 80386DX, 80486 și Pentium Over-drive au 32 biți; începând cu procesoarele Pentium și Pentium Pro procesoarele actuale au o magistrală de date pe 64 biți. Viitoarele versiuni ale procesoarelor x86 pot avea o magistrală mai mare.

O magistrală de date pe 8 biți nu limitează procesorul la tipuri de date de 8 biți. Pur și simplu înseamnă că procesorul poate accesa un singur octet de date pe ciclu de acces la memorie. Prin urmare, magistrala de 8 biți de pe un 8088 poate transmite doar jumătate din informație în unitatea de timp (ciclu de memorie) ca magistrala de 16 biți pe 8086. Prin urmare, procesoarele cu o magistrală de 16 biți sunt în mod natural mai rapide decât procesoarele pe 8 biți. Dimensiunea magistralei de date afectează performanțele sistemului mai mult decât dimensiunea

1 I/O – din engl. Input/Output

oricărei alte magistrale.

Deși există o mică controversă cu privire la „dimensiunea” unui procesor, majoritatea specialiștilor consideră că numărul de linii de date ale unui procesor determină dimensiunea acestuia. Deoarece magistralele de date ale familiei x86 au 8, 16, 32 sau 64 biți lățime, accesul la date se face de asemenea pe 8, 16, 32 sau 64 biți. Deși este posibil să proceseze date pe 12 biți cu un 8088, majoritatea programatorilor procesează 16 biți, deoarece procesorul va prelua și manipula 16 biți oricum.

Deși membrii 64 biți ai familiei x86-64 pot prelucra date până la lățimea magistralei, aceștia pot accesa, de asemenea, unități de memorie mai mici de 8, 16 sau 32 biți. Prin urmare, orice se poate realiza cu o magistrală de date mai mică se poate realiza și cu o magistrală de date mai mare; cu toate acestea, magistrala de date mai mare poate accesa memoria mai rapid și poate accesa blocuri mai mari de date într-o singură operație de transfer din/în memorie.

2.2.1.2. Magistrala de adrese

Magistrala de date a procesoarelor din familia x86 transferă informații între o anumită locație de memorie sau dispozitiv I/O și procesor. Pentru a diferenția locațiile de memorie și dispozitivele I/O, proiectantul de sistem alocă o adresă de memorie unică fiecărui element de memorie și dispozitiv I/O. Când software-ul dorește să acceseze o anumită locație de memorie sau un dispozitiv I/O, plasează adresa corespunzătoare pe magistrala de adrese. Circuitul asociat cu dispozitivul de memorie sau I/O recunoaște această adresă și preia (citește) sau depune (scrie) date pe magistrala de date, toate celelalte locații de memorie sau dispozitive I/O ignoră solicitarea. Doar dispozitivul a cărui adresă se potrivește cu valoarea de pe magistrala de adrese răspunde cererii.

Pe o magistrală pe `n` biți, procesorul poate furniza un total de 2^n adrese unice. Prin urmare, numărul de biți din magistrala de adrese va determina numărul maxim de memorii adresabile și locații de I/O. Procesorul 8086, de exemplu, are o magistrală de adrese pe 20 biți. Prin urmare, acesta poate accesa până la 1 048 576 (sau 2^{20}) locații de memorie.

Procesoarele din familia x86-32 au o magistrală de adrese de 32 biți, astfel acestea pot adresa un spațiu de memorie de 2^{32}=4 294 976 296 locații (octeți), adică 5 GiB.

Procesoarele din familia x86-64 au o magistrală de adrese de 64 biți, deci, teoretic, vor putea adresa un spațiu de memorie de 2^{64} locații (octeți). Deși adresele virtuale au 64 biți lățime în modul 64 biți, implementările actuale (și toate cipurile despre care se știe că se află în etapele de planificare) nu permit utilizarea întregului spațiu de adrese virtuale de 2^{64} de octeți (16 EiB). Aceasta ar fi de aproximativ patru miliarde de ori mai mare decât spațiul de adrese virtuale pe mașini pe 32 biți. Majoritatea sistemelor de operare și aplicațiilor nu vor avea nevoie de un spațiu de adrese atât de mare în viitorul apropiat, astfel încât implementarea unor astfel de adrese virtuale largi ar crește pur și simplu nejustificat complexitatea și costul traducerii adreselor fără nici un beneficiu real. Prin urmare, AMD a decis că, în primele implementări ale arhitecturii, numai cei mai nesemnificativi 48 biți ai unei adrese virtuale vor fi de fapt utilizate în translarea adreselor [4].

În plus, specificația AMD necesită ca cei mai semnificați 16 biți ai oricărei adrese virtuale (biții 48 până la 63), să fie copii ale bitului 47 (într-un mod asemănător cu extensia cu semn a numerelor). Dacă această cerință nu este îndeplinită, procesorul va genera o excepție [4]. Adresele care respectă această regulă sunt denumite „forme canonice”. A adresele în formă canonică sunt cele cuprinse în intervalele `0000000000000000h-00007FFFFFFFFFFFh` respectiv `FFFF800000000000h-FFFFFFFFFFFFFFFFh`, pentru un total de 256 TiB spațiu de adrese virtuale utilizabil. Acest lucru este încă 65536 (64Ki) de ori mai mare decât spațiul virtual de adrese de

4 GiB al mașinilor pe 32 biți.

„Dimensiunea” unui procesor nu are nici o legătură cu magistrala de adrese. Dacă un procesor este pe 64 biți, aceasta înseamnă că magistrala de date este de 64 biți (și are regiștrii de uz general pe un 64 biți).

Din punct de vedere software (mai ales pentru sistemele de operare), expresia „x64” sau „pe 64 biți” înseamnă că programul (sau sistemul de operare) poate utiliza spațiul de adrese de memorie virtuală pe 64 biți. Astfel o adresă virtuală poate avea oricare dintre cele 2^{64} valori distincte (2^{64} = 16Ei = 18 446 744 073 709 551 616).

2.2.1.3. Magistrala de control

Magistrala de control este o colecție de semnale electrice care controlează modul în care procesorul comunică cu restul modulelor sistemului de calcul. Pentru a specifica, de exemplu, direcția fluxului de date pe magistrala de date, există două linii pe magistrala de control, *read* și *write*, care specifică dacă se va efectua citire sau scriere a datelor. Alte semnale includ semnale de *clock*, linii de întrerupere, linii de stare ș.a.m.d.. Alcătuirea exactă a magistralei de control variază în funcție de modelul procesorului din familia 80x86. Cu toate acestea, unele linii de control sunt comune tuturor procesoarelor.

Liniile de control de *read* ($\overline{\texttt{RD}}$) și *write* ($\overline{\texttt{WR}}$), sunt active pe `0` și controlează direcția datelor vehiculate pe magistrala de date. Când ambele au valoarea `1` ($\overline{\texttt{RD}}$=$\overline{\texttt{WR}}$=1), procesorul nu comunică cu memoria și perifericele. Dacă $\overline{\texttt{WR}}$=0 și $\overline{\texttt{RD}}$=1, procesorul citește date din memorie/periferic (adică sistemul transferă date din memorie/periferic către procesor). Dacă $\overline{\texttt{RD}}$=0 și $\overline{\texttt{WR}}$=1, sistemul transferă datele de la procesor către o locație de memorie sau periferic.

Liniile de activare a *bank*-urilor sunt un alt set de linii de control importante. Aceste linii de control permit procesoarelor pe 64 biți să opereze cu blocuri de date mai mici (mai multe detalii în secțiunea 2.2.2).

Familia de procesoare x86, spre deosebire de multe alte procesoare, oferă două spații de adrese distincte: unul pentru memorie și unul pentru periferice (I/O). În timp ce magistrala de adrese pentru memorie pe diverse procesoare x86 variază ca dimensiune, magistrala de adrese I/O are pe toate procesoarele x86 are o lățime de 16 biți. Acest lucru permite procesorului să adreseze până la 65 536 de locații de I/O diferite. După cum se dovedește, majoritatea dispozitivelor (cum ar fi tastatura, imprimanta, mouse-ul, etc.) necesită mai multe locații de I/O. Cu toate acestea, 65 536 de locații sunt mai mult decât suficiente pentru majoritatea aplicațiilor. Designul original al computerului IBM a permis doar utilizarea a 1024 dintre acestea.

Deși familia de procesoare x86 acceptă două spații de adrese, nu are două magistrale diferite (pentru I/O și memorie). În schimb, sistemul partajează magistrala de adrese atât pentru periferice cât și pentru adrese de memorie. Liniile de control suplimentare specifică dacă adresa este destinată memoriei sau zonei I/O. Când aceste semnale sunt active, dispozitivele I/O folosesc adresa de pe cei mai nesemnificativi 16 biți ai magistralei de adrese. Când sunt inactive, dispozitivele I/O ignoră semnalele de pe magistrala de adrese.

2.2.2. Memoria sistemului

Un procesor x86-64 are o magistrală de adrese de 64 biți, deci poate adresa maxim 264 locații de memorie diferite. Procesoarele x86-64 acceptă o memorie adresabilă la nivel de octet (*byte*). Prin urmare, unitatea de bază de memorie este octetul. Deci, cu 64 de linii de adrese, procesoarele x86-64 pot adresa 16 EiB de memorie.

Memoria poate fi văzută ca un vector de octeți. Adresa primului octet este `0` și adresa ultimului octet este 2^{64}-1. Astfel pentru procesoarele x86-64 următoarea declarație de tip

pseudo-C reprezintă o bună aproximare a memoriei:

```
char Memory[18446744073709551616];
```

Pentru a citi un octet din memorie procesorul depune mai întâi pe magistrala de date valoarea dorită (a fi scrisă), apoi pe magistrala de adrese va depune valoarea locației dorite, după care va activa semnalul de *read* pentru a trimite comanda de citire.

Pentru a scrie un octet în memorie procesorul depune mai întâi pe magistrala de adrese va depune valoarea locației dorite, va activa semnalul de *write* pentru a trimite comanda de scriere și abia apoi va prelua data dorită de pe magistrala de date.

Pentru a accesa valori mai mari de 8 biți, diferite sisteme de calcul oferă au implementări diferite. Familia x86 de procesoare stochează cel mai nesemnificativ octet la adresa specificată și restul octeților până la cel mai semnificativ la adresele imediat următoare. Această schemă de reprezentare a numerelor pe mai mulți octeți se numește, și deoarece octetul cel mai nesemnificativ este memorat primul, octeții constituenți ai unui număr pe mai mult de 8 biți (un octet) sunt memorați în ordine inversă. Avantajele schemei *little-endian* sunt:

- octetul cel mai nesemnificativ este întotdeauna stocat la aceeași adresă, indiferent de dimensiunea tipului de date folosit;
- facilitează efectuarea operațiilor aritmetice, deoarece majoritatea operațiilor aritmetice se efectuează începând cu partea mai nesemnificativă către partea cea mai semnificativă (operanzii se parcurg de la cea mai mică către cea mai mare adresă).

Sistemul complementar lui *little-endian* se numește *big-endian*, schemă în care cel mai semnificativ octet este salvat primul. Această metodă oferă avantajul că datele nu trebuie prelucrate în vederea transmiterii acestora prin rețea. Valorile transmise prin rețea sunt reprezentate în formatul *network byte order*, care este același cu formatul *big-endian*.

Astfel este posibil ca, în schema *little-endian*, valorile de octet, cuvânt, dublu-cuvânt, cvadruplu-cuvânt și octa-cuvânt să se suprapună în memorie (a se vedea Figura 2.2).

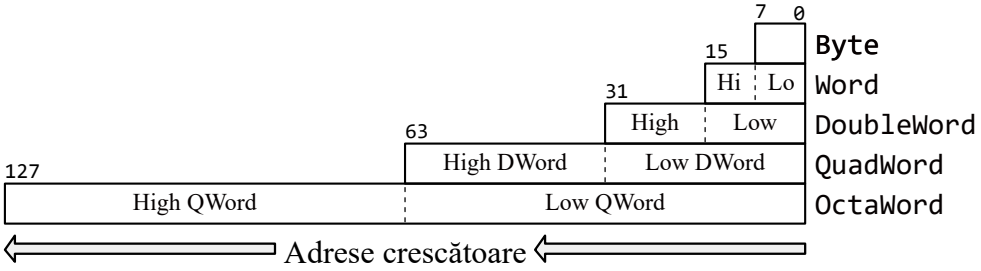

Figura 2.2. Stocarea tipurilor de date în memorie.

Sintagma „memorie adresabilă la nivel de octet” înseamnă că procesorul poate adresa memoria în pachete de măcar un singur octet. Înseamnă, de asemenea, că aceasta este cea mai mică unitate de memorie care poate fi accesată simultan de către procesor. Adică, dacă procesorul dorește să acceseze o valoare de 4 biți, trebuie să citească 8 biți și apoi să ignore cei 4 biți în plus.

Pentru a asigura accesul simultan la serii consecutive de adrese, arhitectura x86 folosește o arhitectură a sistemului de memorie organizată pe *bank*-uri. Sistemul de memorie este compus din mai multe circuite de memorie independente, de dimensiune egală. Acestea sunt numite *bank*-uri și sunt în sistemele x86-64 în număr de 8 (un număr de 64 biți este compus din 8 octeți) și sunt numerotate de la `0` la `7`. În Figura 2.3 se poate observa organizarea pe *bank*-uri a memoriei sistem pentru procesoarele pe 64 biți din familia x86-64.

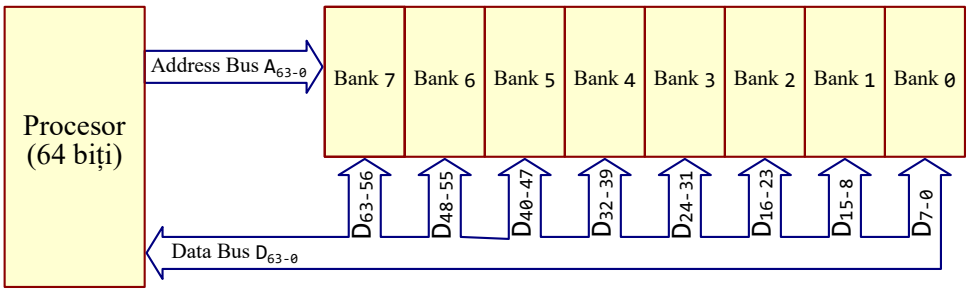

Figura 2.3. Organizarea memoriei pe *bank*-uri la procesoarele pe 64 biți.

Procesoarele pe 64 biți din familia x86-64 vor transfera datele pe liniile de date corespunzătoare funcție de valoarea adresei. De exemplu, pentru a accesa un octet de la o adesă pentru care restul împărțirii valorii adresei la 8 (`Address MOD 8`) este `5`, valoarea de la adresa selectată va fi transferată (citire sau scriere) pe magistrala de date din/în *bank*-ul `5` pe liniile de date D_{40-47}. Procesorul va activa *bank*-ul corespunzător folosind o linie de control „*bank enable*" sau „*byte enable*". În cele din urmă, procesorul va schimba pozițiile acestor biți pe magistrala de date internă și acestea vor ajunge pe liniile D_{0-7}.

Dacă, de exemplu, se accesează o locație pe 16 biți (2 octeți) de la o adresă pentru care restul împărțirii valorii adresei la 8 (`Address MOD 8`) este `7` se poate observa din Figura 2.4, că valoarea (compusă din doi octeți) nu va găsi la adrese aliniate și transferul se va efectua în doi pași, procesorul va compune automat data în formatul dorit, dar operația va dura mai mult timp. Prin aranjarea cu atenție a modului în care memoria este utilizată, se poate îmbunătăți viteza programelor.

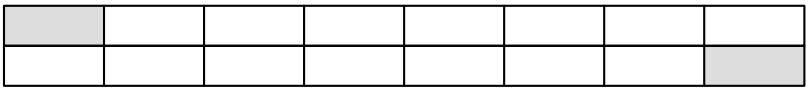

Figura 2.4. Accesul unei locații pe 16 biți de la o adresă (ADDRESS MOD 8) = 7.

În ceea ce privește încărcarea corectă a datelor procesorul gestionează automat accesul. Cu toate acestea, există un beneficiu de performanță în cazul în care datele sunt aliniate corect. Se obține astfel un spor de performanță dacă:

- valorile pe 16 biți (`WORD`) sunt plasate la adrese pare (aliniere la 2 octeți);
- valorile pe 32 biți (`DWORD`) sunt la adrese multiplu de 4 (aliniere la 4 octeți);
- valorile pe 64 biți (`QWORD`) sunt la adrese multiplu de 8 (aliniere la 8 octeți);
- valorile pe 128 biți (`OWORD`) sunt la adrese multiplu de 16 (aliniere la 16 octeți).

2.2.3. Semnalul de clock

Prin semnalul de *clock* se realizează sincronizarea în cadrul unui sistem informatic. Semnalul de *clock* este un semnal electric al magistralei de control care alternează periodic între `0` și `1` (ca în Figura 2.5). Procesorul este un bun exemplu de sistem logic complex sincron.

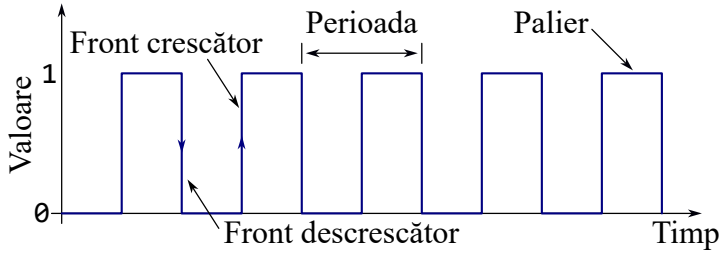

Figura 2.5. Semnal de *clock*.

Frecvența cu care semnalul de *clock* alternează între `0` și `1` este *frecvența de clock* a

sistemului. Timpul necesar pentru ca semnalul de *clock* să treacă de la `0` și `1` și înapoi la `0` este *perioada de clock*. O perioadă completă se mai numește *ciclu de clock*. Frecvența de ceas este pur și simplu numărul de cicluri de *clock* care apar în fiecare secundă.

Pentru a asigura sincronizarea, majoritatea procesoarelor încep o operație fie pe *frontul crescător* (când ceasul trece de la `1` la `0`), fie pe *frontul descrescător* (când ceasul trece de la `0` la `1`). Ceasul sistemului își petrece cea mai mare parte a timpului fie la `0`, fie la `1` și foarte puțin timp între cele două. De aceea fronturile *clock*-ului sunt puncte perfect de sincronizare.

Deoarece toate operațiunile CPU sunt sincronizate în jurul ceasului, CPU nu poate efectua sarcini mai repede decât *clock*-ul (dacă sincronizarea se face pe unul dintre fronturi) sau dublul *clock*-ului (dacă sincronizarea se face pe ambele fronturi). Doar pentru că un procesor funcționează la o frecvență de *clock* nu înseamnă că executa o operație la fiecare ciclu de *clock*. Multe operații iau mai multe cicluri de *clock* pentru a se finaliza, astfel încât procesorul efectuează operațiuni într-un ritm semnificativ mai mic decât frecvența de *clock*.

2.2.4. Temporizarea accesul la memorie

Accesul la memorie este probabil cea mai frecventă activitate a procesorului. Accesul la memorie este cu siguranță o operație sincronizată pe baza semnalului de *clock*. Adică, citirea unei valori din memorie sau scrierea unei valori în memorie are loc nu mai des decât o dată la fiecare jumătate de ciclu de *clock*. Într-adevăr, pe multe procesoare, este nevoie de mai multe cicluri de *clock* pentru a accesa o locație de memorie. Timpul de acces la memorie se măsoară de fapt în numărul de cicluri de *clock* pe care sistemul necesită pentru a accesa o locație de memorie; aceasta este o valoare importantă, deoarece timpii mai lungi de acces la memorie au ca rezultat o performanță mai scăzută.

Timpul de acces la citirea din memorie este perioada de timp din momentul în care procesorul plasează o adresă pe magistrala de adrese și preia datele de pe magistrala de date. Dacă subsistemul de memorie nu funcționează suficient de rapid, procesorul va citi date eronate de pe magistrala de date sau nu va stoca corect datele la o operațiune de scriere a memoriei.

Deoarece timpii de acces la memorie sunt mai mari decât perioada de *clock* (sau frecvența memoriei este mai mică decât cea a procesorului), accesul la memorie nu se poate face într-un singur ciclu de *clock* (cum se întâmpla cazul sistemelor 80486, de exemplu). Pentru a se realiza sincronizarea între memorie și procesor se introduc stări de *wait*.

O stare de *wait* (așteptare) nu este nimic altceva decât un ciclu de *clock* suplimentar pentru a oferi timp dispozitivului să finalizeze o operație. Aproape toate procesoarele de uz general existente oferă un semnal pe magistrala de control pentru a permite introducerea stărilor de *wait*. În general, circuitul de decodare activează această linie pentru a întârzia o perioadă suplimentară de *clock*, dacă este necesar. Dacă o singură stare de *wait* nu este suficientă se pot introduce mai multe.

Evident că, din punct de vedere al performanței sistemului, stările de *wait* nu sunt un lucru bun. În timp ce procesorul așteaptă date din memorie, acesta nu poate opera pe datele respective. Adăugarea unei singure stări de *wait* la un ciclu de memorie pe un întârzie accesarea datelor. Aceasta, la rândul său, reduce la viteza de acces la memorie și indirect numărul de operații pe care procesorul le poate efectua într-o secundă.

2.2.5. Memorie cache

În urma analizei cercetătorilor asupra programelor tipice, s-a observat că acestea tind să acceseze în mod repetat aceleași locații de memorie. Ba mai program accesează adesea locații adiacente de memorie. Pentru a exploata aceste fenomene apărut memoria *cache*.

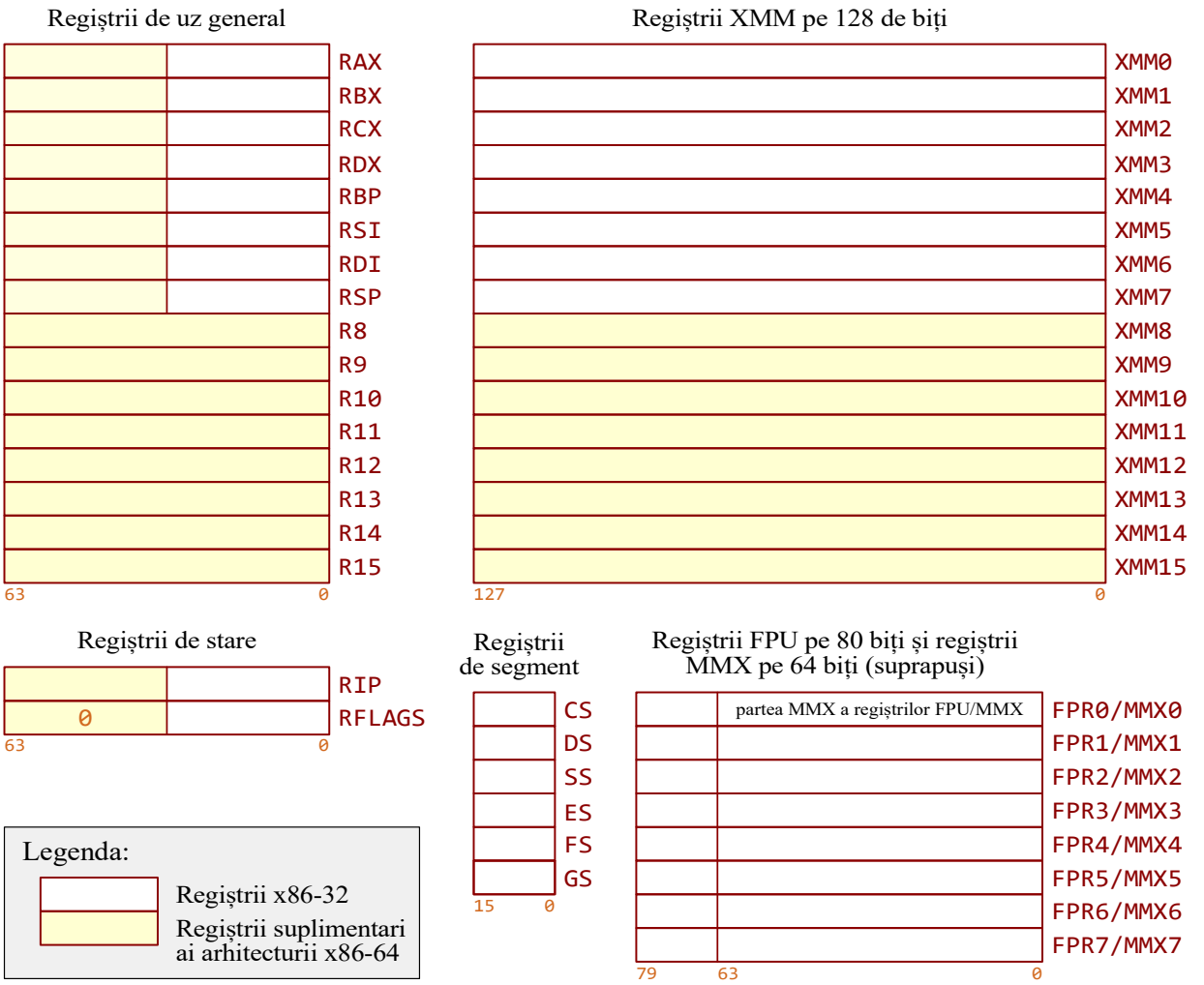

Figura 2.6. Setul de regiștri x86-64.

Memoria cache este un bloc intermediar între procesor și memoria principală. Este o cantitate mică de memorie foarte rapidă (cu zero stări de *wait*). Spre deosebire de memoria normală, octeții care apar într-un cache nu au adrese fixe. În schimb, memoria cache poate reasigna adresa locații de date. Aceasta permite sistemului să păstreze valorile accesate recent în *cache*. Adresele la care procesorul nu le-a accesat niciodată sau nu au fost accesate o perioadă de timp rămân în memoria principală (lentă). Deoarece majoritatea acceselor la memorie sunt la variabilele accesate recent (sau în locații din apropierea unei locații accesate recent), aceste date apar în general în memoria cache.

Memoria cache nu este perfectă. Deși un program poate „petrece” mult timp executând o secvență de cod, în cele din urmă va apela o procedură sau va sări la o secțiune de cod din afara memoriei *cache*. În acest caz, procesorul trebuie să acceseze memoria principală pentru a prelua datele. Deoarece memoria principală este lentă, aceasta va necesita introducerea stărilor de *wait*.

Atunci când procesorul accesează memoria și găsește datele în cache are loc un așa numit „*cache hit*” în memoria *cache*. Într-un astfel de caz, procesorul poate accesa de obicei date fără stări de *wait*. Similar un „*cache miss*” are loc dacă procesorul accesează memoria și dacă datele nu sunt prezente în *cache*, și acesta trebuie să citească datele din memoria principală, suferind o pierdere de performanță. Procesorul copiază datele în cache ori de câte ori este accesată o adresă care nu este prezentă în *cache*, deoarece este probabil ca sistemul să acceseze aceeași locație în scurt timp.

Memoria *cache* nu gestionează doar aspectele temporale ale accesului la memorie, ci și aspectele spațiale. Pentru a rezolva problema accesului la locații consecutive, majoritatea sistemelor de memorie în *cache* citesc mai mulți octeți consecutivi din memorie atunci când apare un *miss* în *cache*. Majoritatea cipurilor de memorie moderne au moduri speciale care să permită accesul rapid mai multe locații consecutive de memorie. *Cache*-ul exploatează această capacitate pentru a reduce numărul mediu de stări de *wait* necesare pentru a accesa memoria.

Evident că că raportul dintre *hit*-uri și *miss*-uri crește odată cu dimensiunea (în octeți) a sub-sistemului de memorie *cache*. Un alt mod de a îmbunătăți performanța este implementarea unui sistem de *cache* pe două niveluri. Dacă primul nivel (nivelul 1) este o memoria rapidă fără stări de *wait* (dar costisitoare), al doilea nivel (nivelul 2) este un cache secundar, de capacitate mai mare, situat între memoria cache de nivel 1 și memoria principală. Majoritatea proiectanților de sistem utilizează o memorie mai lentă care necesită una sau două stări de *wait*. Aceasta este încă mult mai rapidă decât memoria principală. În combinație cu memoria cache de nivel 1, se pot obține performanțe mai bune din partea sistemului. Similar se pot adăuga teoretic oricâte niveluri de memorie cache, fiecare nivel superior având o capacitate mai mare, dar având o viteză mai mică decât cel anterior, memoria sistem fiind situată pe ultimul nivel.

2.2.6. Regiștrii procesorului

Regiștrii procesorului sunt locații de memorie foarte speciale construite din bistabile. Acestea nu fac parte din memoria principală și sunt implementate direct în cipul procesorului. Diverși membri ai familiei x86 au dimensiuni diferite ale regiștrilor.

Deoarece regiștrii sunt situați direct în procesor și sunt gestionați de către acesta, acestea sunt mult mai rapide decât memoria. Accesarea unei locații de memorie necesită mai multe cicluri de *clock*. Accesarea datelor dintr-un registru durează zero (`0`) cicluri de *clock*. Prin urmare, se recomandă păstrarea variabilelor în regiștri. Seturile de regiștri sunt foarte mici, iar o parte a acestora au scopuri speciale care limitează utilizarea lor ca variabile, dar sunt încă un loc excelent pentru a stoca temporar date.

În DEX[1] cuvântul *registru* este de gen neutru, astfel încât forma de plural ar fi *registre*, în schimb este menționată o formă de gen masculin (*regiștri*) cu semnificația „21. s.m. (inf) zonă de memorie din interiorul microprocesorului utilizată la stocarea valorilor și adreselor memoriei externe în timp ce microprocesorul execută operații logice sau aritmetice asupra lor”. Există mai multe discuții legate de variațiile unor substantive între formele de gen neutru și cele de gen masculin. În principiu, publicul larg utilizează formele neutre: *registre*, *rezistoare* sau *lasere*, de exemplu. În literatura de specialitate sunt preferate în schimb formele masculine: *regiștri*, *rezistori* sau *laseri*. Una din explicații ar putea fi aceea că formele masculine sugerează obiecte concrete, în timp ce formele neutre mai degrabă noțiuni abstracte. O altă explicație ar fi că variantele masculine sunt de obicei mai scurte cu o silabă și sunt astfel, mai compacte și mai apropiate de forma din limbi de circulație internațională. Un exemplu similar este cuvântul *virus*, care păstrează forma de neutru pentru *virusuri* biologice, și forma de masculin pentru *viruși* informatici.

2.3. Regiștrii arhitecturii x86-64

Arhitectura x86-64 este o extensie simplă, dar puternică, pe 64 biți, compatibilă cu arhitectura x86 standard. Acesta adaugă adresarea pe 64 biți și extinde resursele de tip registru pentru a sprijini performanțe superioare pentru programele pe 64 biți. Aplicațiile pe 64 biți beneficiază astfel, atât de adrese pe 64 biți, cât și de un număr crescut de regiștri. Numărul redus

1 DEX este Dicționarul EXplicativ al limbii române, care conține termeni și definiții în limba română.

de regiștri disponibili în arhitectura x86-32 limitează performanța în aplicații intense de calcul. Creșterea numărului de regiștri oferă un spor de performanță pentru multe tipuri aplicații.

După cum se poate observa și în Figura 2.6 modificările aduse regiștrilor arhitecturii x86-64 față de generația anterioară (x86-32) constau în:

- 8 regiștri de uz general suplimentari (GPR[1]);
- extensia tuturor regiștrilor de uz general și a celor de stare la 64 biți;
- 8 regiștri suplimentare YMM/XMM.
- adresare uniformă a regiștrilor la nivel de octet pentru toate GPR-urile.

În total arhitectura x86-64 are:

- 16 regiștri de uz general pe 64 biți;
- 2 regiștri de stare pe 64 biți;
- 6 regiștri de segment pe 16 biți;
- 8 regiștri FPU pe 80 biți suprapuși cu cei 8 regiștri `MMX`[2] pe 64 biți;
- 16 regiștri `XMM` pe 128 biți suprapuși cu regiștrii `YMM` pe 256 biți.

2.3.1. Regiștrii de uz general

Regiștrii de uz general se pot împărți în trei categorii. Patru dintre aceștia, `"A"`, `"B"`, `"C"` și `"D"`, sunt clasificați ca regiștri de date. Acești regiștri de date sunt accesibili fie ca registru complet pe 64 biți (`QWORD`), reprezentat de prefixul `R` și sufixul `X`, jumătatea mai nesemnificativă a acestora pe 32 biți (`DWORD`), reprezentat de prefixul `E` și sufixul `X`, sfertul mai nesemnificativ pe 16 biți (`WORD`), fără prefix cu sufixul `X`, octetul mai nesemnificativ, fără prefix cu sufix `L`, și al doilea octet, desemnat cu sufixul `H` (structura se poate observa în Figura 2.7). Registrul `RAX`, de exemplu, poate fi accesat pe 32 biți (cei mai nesemnificativi) ca `EAX`, pe 16 biți ca `AX` și pe 8 biți ca `AL`. Octetul mai semnificativ al registrului `AX` poate fi accesat ca `AH`. Regiștrii `RAX`, `RBX`, `RCX` și `RDX` sunt singurii pentru care poate fi accesat al doilea octet (biții 8-15) ca `AH`, `BH`, `CH` respectiv `DH`.

Acești regiștri sunt numiți regiștri de uz general deoarece pot fi utilizați pentru toate tipurile de operații: pot conține date, adrese de memorie a unei operații (*pointer*) sau pot fi utilizați pentru calculul adreselor de memorie. Fiecare dintre acești regiștri au una sau mai multe funcții suplimentare implicite:

- `RAX`, care este numit registru acumulator, este utilizat pentru toate operațiunile de intrare/ieșire și unele operații aritmetice (de exemplu, înmulțire și împărțire);
- `RBX`, care este denumit registrul de bază, poate fi folosit ca registru de adrese.
- `RCX`, care este registrul de contorizare, este folosit de instrucțiuni care necesită numărare (de obicei este utilizat pentru controlul numărului de repetări ale unei bucle și în operații de deplasare a biților);
- `RDX`, care este registrul de date, este folosit pentru unele operații de intrare/ieșire și, de asemenea, la înmulțire și împărțire.

1 GPR(s) – din engl. General-Purpose Register(s).
2

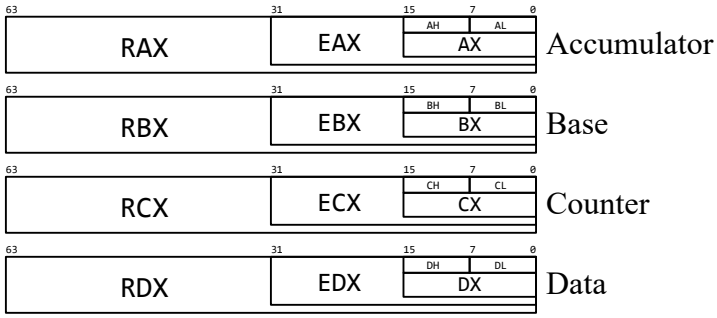

Figura 2.7. Structura regiștrilor RAX, RBX, RCX și RDX.

A doua categorie de regiștri sunt registrele de adrese/index. Aceasta includ următoarele patru registre: Stack Pointer (SP), Base Pointer (BP), Source Index (SI) și Destination Index (DI). Aceștia sunt accesibili fie ca registri compleți pe 64 biți (QWORD), reprezentat de prefixul R, jumătatea mai nesemnificativă a acestora pe 32 biți (DWORD), desemnat de prefixul E, sfertul mai nesemnificativ pe 16 biți (WORD), fără prefix sau sufix și octetul cel mai nesemnificativ, fără prefix cu sufix L (a se vedea Figura 2.8).

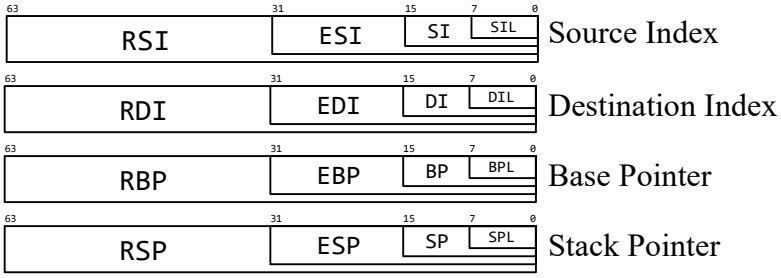

Figura 2.8. Structura regiștrilor RSI, RDI, RBP și RSP

Acești patru regiștri de uz general vin cu câteva îndrumări pentru utilizarea lor implicită. Instrucțiunile cu operanzi impliciți, adică operanzi care se presupun a utiliza un anumit registru și, prin urmare, nu necesită codificarea acelui operand, și permit astfel codificări mai scurte pentru utilizări comune. Aceste utilizările sunt următoarele:

- regiștrii RSI și RDI sunt utilizați implicit în instrucțiunile x86 cu șiruri;
- registrul RBP este utilizat ca pointer la datele salvate pe stivă și este utilizat în general pentru a accesa parametrii și variabilele locale unei proceduri sau funcții;
- registrul RSP are un rol foarte important - menține stiva programelor; în mod normal, acest registru nu este utilizat pentru calcule aritmetice deoarece funcționarea corectă a majorității programelor depinde de utilizarea atentă a acestui registru.

Cea de-a treia categorie de regiștri de uz general o constituie noii regiștri adăugați de arhitectura x86-64. Aceștia sub regiștri de uz general și nu au alte utilizări speciale din punct de vedere al instrucțiunilor procesorului. Acești regiștri sunt în număr de 8, sunt pe 64 biți și sunt numerotați de la R8 până la R15. Pentru a accesa acești regiștri pe 32 biți se adaugă sufixul D (de la DWORD), pentru accesul pe 16 biți sufixul W (de la WORD) și pentru accesul pe 8 biți sufixul B (de la BYTE), după cum se poate observa în Figura 2.9.

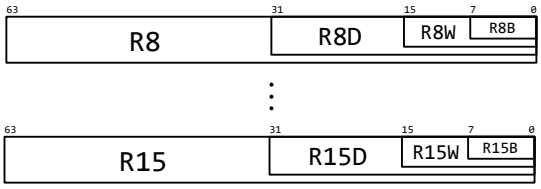

Figura 2.9. Structura regiștrilor R8 – R15.

Utilizarea implicită a diferiților regiștrilor de uz general oferă un ajutor programatorului, care, odată familiarizat cu diferitele semnificații ale acestora, va putea înțelege mai ușor și mai rapid un algoritm în asamblare, presupunând că acesta se conformează convențiilor. Acest lucru este similar, într-o oarecare măsură, modului în care numele variabilelor ajută programatorul să înțeleagă rolul lor. Este important de reținut că acestea sunt doar indicii (sugestii), nu reguli în sine.

Pe lângă regiștrii de uz general procesorul are doi regiștri de stare: `RIP` (Instrucțion Pointer) și `RFLAGS` – registrul indicatorilor de stare.

2.3.2. Registrul RIP

Registrul de 64 biți `RIP` (Instruction Pointer), uneori referit și ca PC (Program Counter), conține adresa din zona curentă de cod a următoarei instrucțiuni care urmează a fi executată. Este avansat de la o limită a unei instrucțiuni la alta în execuția secvențială sau este deplasat înainte sau înapoi cu o serie de instrucțiuni atunci când se execută instrucțiuni `JMP`, `Jcc`, `CALL` și `RET`. Modul pe 64 biți acceptă și o tehnică numită adresare relativă `RIP`. Folosind această tehnică, adresa efectivă este determinată prin adăugarea unui deplasament la valoarea din `RIP` (adresa instrucțiunii următoare).

2.3.3. Registrul RFLAGS

În Figura 2.10 se poate observa structura registrului `EFLAGS` care este partea mai nesemnificativă pe 32 biți a registrului `RFLAGS` pe 64 biți.

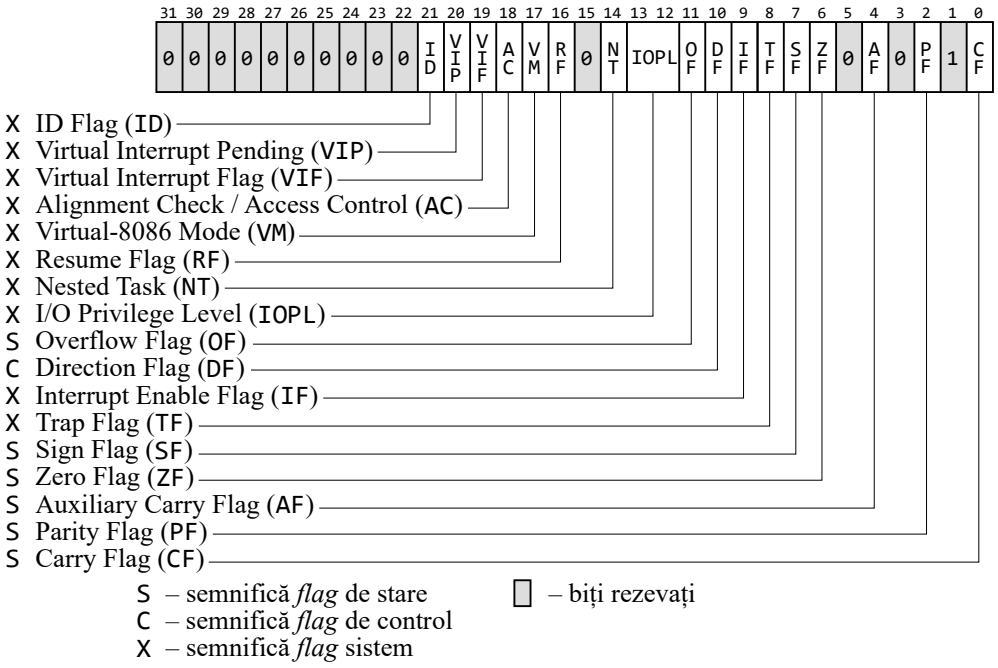

Figura 2.10. Structura registrului `EFLAGS` pe 32 biți.

Acești biți sunt setați de diferite instrucțiuni, de obicei instrucțiuni aritmetice sau logice, pentru a semnala anumite condiții. Aceste *flag*-uri de condiție pot fi ulterior verificate pentru a lua decizii. Dintre toate aceste *flag*-uri *carry* (`CF`), *parity* (`PF`), *zero* (`ZF`), *sign* (`SF`) și *overflow* (`OF`) sunt mai deosebite, deoarece acestea pot fi testate (dacă au valoarea `0` sau `1`) cu instrucțiunile `SETcc` sau cu instrucțiunile de salt condiționat.

Instrucțiunile aritmetice, logice și diverse afectează *flag*-ul *overflow* (`OF`). După o operație aritmetică, acest *flag* conține `1` dacă rezultatul nu se poate reprezenta pe numărul de biți ai operandului destinație în reprezentare cu semn. Dacă rezultatul operației aritmetice cu semn nu produce o depășire, procesorul șterge (resetează la `0`) acest *flag*. Deoarece operațiunile logice se aplică, în general, la valori fără semn, instrucțiunile logice pur și simplu șterg *flag*-ul *overflow*. Alte instrucțiuni lasă *flag*-ul *overflow* pe o valoare arbitrară.

Instrucțiunile cu șirului folosesc *flag*-ul de direcție (`DF` – *direction flag*). Când *flag*-ul *direction* este pe valoarea `0`, elemente șirurilor sunt procesate de la adrese mici (inferioare) către adrese înalte (superioare); iar când este setat pe valoarea `1`, elemente șirurilor sunt procesate în direcția opusă. Pentru mai multe detalii poate fi consultată secțiunea 4.10.

Dacă rezultatul unor calcule este negativ, procesorul setează *flag*-ul de semn (*sign flag* – `SF`). *Flag*-ul SF poate fi testat după o operație aritmetică pentru a verifica dacă rezultatul este. O valoare numerică cu semn este negativă dacă cel mai semnificativ bit este `1` și prin urmare, operațiunile cu valori fără semn vor seta *flag*-ul `SF` dacă rezultatul (pozitiv) are 1 pe poziția cea mai semnificativă.

Diverse instrucțiuni *flag*-ul *zero* (`ZF`) atunci când generează un rezultat zero. Adesea acest *flag* este folosit pentru a verifica dacă două valori sunt egale (de exemplu, după scăderea a două numere, un rezultatul zero indică faptul că acestea sunt acestea sunt egale). Acest *flag* este de asemenea util după diverse operații logice pentru a vedea dacă un anumit bit într-un registru sau dintr-o locație de memorie conține `0` sau `1`.

Flag-ul *auxiliary carry* (`AF`) este utilizat în operații cu numere zecimale în format BCD[1]. Deoarece majoritatea programelor nu lucrează cu numere în format BCD, acest *flag* este utilizat foarte rar. Procesorul x86 nu oferă instrucțiuni care să permită testarea, setarea sau ștergerea directă a acestui *flag*. Doar instrucțiunile `ADD`, `ADC`, `SUB`, `SBB`, `MUL`, `IMUL`, `DIV`, `IDIV` operează cu acesta.

Flag-ul de paritate (*parity flag* – `PF`) este setat în funcție de paritatea celor mai nesemnificativi 8 biți ai rezultatului unei operații. Dacă o operație produce un număr par de biți de `1`, `PF` este setat pe `1`. Acest *flag* este șters (resetat la `0`) dacă operația produce un număr impar de biți de `1`. Acest steag este util în anumite programe de comunicații de date, și are. Mai degrabă, rolul de a oferi o oarecare compatibilitate cu procesorul 8080.

Flag-ul *carry* (`CF`) are mai multe scopuri. În primul rând, denotă o depășire în reprezentarea fără semn (similar cu *flag*-ul *overflow* pentru depășirea pentru numerele cu semn). Acesta poate fi folosit și în timpul operațiilor aritmetice și logice multi-precizie. Anumite instrucțiuni de testare, setare, ștergere și inversare a biților afectează direct acest *flag*. În cele din urmă, având în vedere că `CF` poate fi șters, setat, inversat și testat cu ușurință, este util pentru diverse operații logice booleene.

2.3.4. Regiștrii FPU de date

Co-procesoarele 80x87 (FPU) furnizează opt (8) regiștri de date de 80 biți organizați sub forma unei stive (circulare). Aceasta este o diferență semnificativă de la organizarea regiștrilor de

1 BCD – din eng. Binary-Coded Decimal, este o clasă de codări binare a numerelor zecimale în care fiecare cifră zecimală este reprezentată de un număr fix de biți, de obicei patru sau opt.

uz general ai procesorului x86 care cuprind un set standard de regiștri. Intel referă acești regiștri ca `ST(0)`, `ST(1)`, …, `ST(7)`. Majoritatea asambloarelor acceptă `ST` ca prescurtare pentru `ST(0)`.

Cea mai mare diferență între setul de registre FPU și setul de registre x86-64 este organizarea sub formă de stivă. Pe 80x87, setul de regiștri constituie o stivă de opt elemente cu valori în virgulă mobilă pe 80 biți (precizie extinsă). `ST(0)` se referă la valoarea din vârful stivei, `ST(1)` se referă la următorul element din stivă și așa mai departe. Multe instrucțiuni în virgulă mobilă depun și extrag numere pe/din stivă; prin urmare, `ST(1)` se va referi la conținutul anterior al `ST(0)` după ce depunerea unei noi valori pe stivă.

2.3.5. Regiștrii MMX

Arhitectura MMX adaugă opt (8) regiștri pe 64 de biți. Instrucțiunile MMX referă acești regiștri ca `MM0`, `MM1`, `MM2`, `MM3`, `MM4`, `MM5`, `MM6` și `MM7`. Acestea sunt strict registre de date, și nu pot fi utilizate utiliza pentru a memora adrese și nu sunt potrivite pentru calcule care implică adrese.

Deși `MM0`...`MM7` apar ca regiștri separați în arhitectura Intel, aceștia sunt de fapt suprapuși regiștrilor FPU (`ST0`...`ST7`). Fiecare dintre cei opt regiștri MMX pe 64 biți este echivalent fizic cu cei mai nesemnificativi 64 biți din fiecare registru FPU. Registrele MMX se suprapun pe registrele FPU în același mod în care registrele de uz general pe 32 biți se suprapun pe registrele de uz general pe 64 biți.

Deoarece registrele MMX se suprapun peste registrele FPU, nu se pot amesteca instrucțiunile FPU și MMX în aceeași secvență de calcul. După execuția unei instrucțiuni MMX, nu se poate executa o altă instrucțiune FPU până când nu se execută o instrucțiune MMX specială, `EMMS` (Exit MMX Machine State).

Intel, în literatura lor, se laudă constant cu privire la ideea grozavă de a suprapune cele două seturi de regiștri. Prin faptul că regiștrii MMX sunt de fapt regiștrii FPU, Microsoft și alți furnizori de sisteme de operare *multitasking* nu au fost nevoiți să scrie cod special pentru a salva starea MMX atunci când procesorul trece de la un proces la altul. Faptul că sistemul de operare salvează automat starea FPU înseamnă că procesorul va salva automat și starea MMX. Acest lucru a însemnat că noile cipuri Pentium cu tehnologia MMX create de Intel la momentul respectiv, erau automat compatibile cu Windows 95, Windows NT și Linux fără a efectua modificări ale codului sistemului de operare.

Regiștrii MMX pot conține doar valori numerice întregi. Fiecare registru are 64 biți și poate fi utilizat pentru a stoca fie numere întregi pe 64 biți, fie mai multe numere întregi mai mici într-un format împachetat: o singură instrucțiune poate fi apoi aplicată la două numere întregi de 32 biți, patru numere întregi de 16 biți sau opt numere întregi de 8 biți simultan.

2.3.6. Regiștrii SSE

Setul de instrucțiuni SSE, introdus de Intel în 1999 odată cu Pentium III, adaugă arhitecturii x86 opt noi regiștri de 128 de biți: `XMM0`...`XMM7`. Inițial, un registru SSE putea fi folosit doar pentru patru numere virgulă mobilă precizie simplă pe 32 de biți (echivalentul unui `float` în C). Setul de instrucțiuni SSE2 a extins capacitățile registrelor XMM, astfel încât acestea pot fi utilizate ca:

- două (2) numere în virgulă mobilă pe 64 biți (precizie dublă);
- două (2) numere întregi pe 64 de biți;
- patru (4) numere în virgulă mobilă pe 32 biți (precizie simplă);
- patru (4) numere întregi pe 32 de biți;
- opt (8) numere întregi pe 16 de biți;
- șaisprezece (16) caractere pe 8 biți (octeți).

Arhitectura x86-64 extinde numărul acestor regiștri cu încă opt pentru un total de 16: referiți de la `XMM0` până la `XMM15`.

2.3.7. Regiștrii AVX

Setul de instrucțiuni AVX[1] sunt extensii la arhitectura setului de instrucțiuni x86 pentru microprocesoare de la Intel și AMD propuse de Intel în martie 2008 și implementate pentru prima dată de Intel pe procesoare cu nuclee Sandy Bridge.

AVX utilizează șaisprezece (16) registre YMM pe 256 biți, notate de la `YMM0` până la `YMM15`. Fiecare registru YMM poate stoca și efectua operațiuni simultane (matematice) pe:

- opt (8) numere în virgulă mobilă pe 32 biți (precizie simplă);
- patru (4) numere în virgulă mobilă pe 64 biți (precizie sublă).

Lățimea regiștrilor SIMD[2] este crescută de la 128 biți la 256 biți și redenumiți de la `XMM0`...`XMM7` la `YMM0`...`YMM7` (în modul x86-64, de la `XMM0`...`XMM15` la `YMM0`...`YMM15`). Instrucțiunile SSE vechi pot fi utilizate în continuare pentru a opera pe cei mai nesemnificativi 128 biți ai regiștrilor YMM.

2.4. Moduri de adresare ale procesorului x86-64

Modurile de adresare reprezintă un aspect al arhitecturii setului de instrucțiuni al unei unități centrale de procesare (CPU). Într-o arhitectură sunt definite diferitele moduri de adresare care reprezintă modalitatea prin care acea arhitectură identifică operandul (sau operanzii) fiecărei instrucțiuni. Un mod de adresare specifică modalitatea de calculare a adresei efective de memorie a unui operand folosind informații păstrate în regiștri și/sau constante și care sunt conținute într-o instrucțiune.

Arhitectura x86-64 are mai multe moduri de adresare:

- adresare imediată (constante pe 8, 16, 32 și 64 biți)
- operand registru (regiștri de 8, 16, 32 și 64 biți)
- deplasament (adresare directă)
- adresare bazată
- adresare bazată cu deplasament
- adresare indexată și scalată cu deplasament
- adresare bazată și indexată cu deplasament
- adresare bazată, indexată și scalată cu deplasament
- adresare relativă (față de `RIP`)

} operanzii sunt în memorie

2.4.1. Operand imediat

Unele instrucțiuni folosesc date codificate în însăși codul instrucțiunii ca operand sursă. Acești operanzi sunt numiți operanzi imediați (sau valoare imediată). De exemplu, următoarea instrucțiune `ADD` adaugă o valoare imediată de `24` conținutului registrului `RAX`:

```
add    rax, 24
```

Toate instrucțiunile aritmetice (cu excepția instrucțiunilor `DIV` și `IDIV`) permit operand sursă cu valoare imediată. Valoarea maximă permisă pentru un operand imediat variază în funcție de instrucțiuni, dar nu poate mai mare decât valoarea maximă a unui număr întreg fără semn pe 32 biți (2^{32}). Excepție face instrucțiunea `MOV` care este singura care permite operand sursă imediat

1 AVX – din engl. Advanced Vector Extensions, cunoscut și sub numele de Sandy Bridge New Extensions.

2 SIMD – din engl. Single Instruction, Multiple Data, descrie procesoare (sau instrucțiuni) cu mai multe elemente de procesare care efectuează aceeași operație pe mai multe date simultan.

pe 64 de biți, operandul destinație poate fi doar registru pe 64 biți în acest caz.

2.4.2. Operanzi registru

Operanzii registru în modul pe 64 de biți poate fi unul dintre:

- regiștri de uz general pe 64 biți: `RAX`, `RBX`, `RCX`, `RDX`, `RSI`, `RDI`, `RSP`, `RBP` sau `R8-R15`;
- regiștri de uz general pe 32 biți: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `ESP`, `EBP` sau `R8D-R15D`;
- regiștri de uz general pe 16 biți: `AX`, `BX`, `CX`, `DX`, `SI`, `DI`, `SP`, `BP` sau `R8W-R15W`;
- regiștri de uz general pe 8 biți: `AL`, `BL`, `CL`, `DL`, `SIL`, `DIL`, `SPL`, `BPL` și `R8L-R15L`;
- regiștri de uz general pe 8 biți: `AH`, `BH`, `CH`, `DH`;
- regiștrii de segment: `CS`, `DS`, `SS`, `ES`, `FS` și `GS`;
- registru `RFLAGS`;
- regiștrii FPU x87: `ST(0)` până la `ST(7)`, cuvintele de stare, control, *tag*, pointer la operandul de date și instrucțiune;
- regiștrii MMX (`MM0` – `MM7`);
- regiștrii XMM (`XMM0` până `XMM15`) și registru MXCSR;
- regiștrii de control (`CR0`, `CR2`, `CR3`, `CR4` și `CR8`) și regiștrii de adrese de tabel de sistem (`GDTR`, `LDTR`, `IDTR` și registrul de sarcini);
- regiștrii de *debug* (`DR0`, `DR1`, `DR2`, `DR3`, `DR6` și `DR7`);
- regiștrii MSR;
- perechea de regiștri `RDX:RAX` reprezentând un operand pe 128 biți.

2.4.3. Adresarea memoriei în arhitectura x86-64

Pentru a accesa o valoare din memorie procesorul trebuie să cunoască adresa acelei locații. Adresa de memorie poate fi specificată direct ca o valoare imediată pe 64 de biți (adresare directă) sau printr-un calcul de adrese format dintr-una sau mai multe dintre următoarele componente:

- deplasament – o valoare imediată pe 8, 16, sau 32 biți.
- baza – valoarea dintr-un registru uz general general;
- index – valoarea dintr-un registru uz general general;
- scală – o valoare de 2, 4 sau 8 care se înmulțește cu valoarea indexului.

Valoarea adresei care rezultă prin sumarea acestor componente se numește adresă efectivă. Fiecare dintre aceste componente poate avea o valoare pozitivă sau negativă (reprezentare în complement față de 2), cu excepția factorului de scalare.

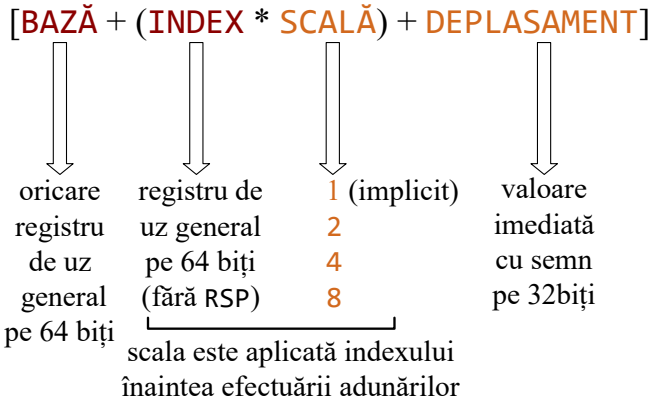

Figura 2.11. Schema de calcul pentru adresa efectivă în adresarea indirectă.

Figura 2.11 prezintă toate modalitățile posibile prin care aceste componente pot fi combinate pentru a crea o adresă efectivă în într-un segment. Utilizarea registrelor de uz general ca bază sau de index este restricționată doar cazul registrului `RSP`, care nu poate fi utilizat ca registru de index.

Componentele calculului adresei efective: baza, indexul și deplasamentul pot fi utilizate în orice combinație, iar oricare dintre aceste componente poate fi nule (zero). Un factor de scară poate fi utilizat numai atunci când este utilizat un index. Fiecare combinație posibilă este utilă pentru structurile de date utilizate frecvent de programatori în limbaje la nivel înalt și limbaj de asamblare.

Următoarele moduri de adresare sugerează utilizări pentru combinații comune de componente de adresă.

- **Directă** – Prezența doar a deplasamentului reprezintă o adresare directă a operandului. Deoarece deplasamentul este codat în instrucțiune, această formă de adresare este uneori numită adresă absolută sau statică. Este utilizată în mod obișnuit pentru a accesa un operand scalar alocat static.
- **Bazată** – O bază reprezintă doar o adresare indirectă simplă a operandului. Deoarece valoarea din registrul de bază se poate modifica, adresarea bazată poate fi utilizată pentru stocarea dinamică a variabilelor și a structurilor de date.
- **Bazată cu deplasament** – Un registru de bază și un deplasament pot fi utilizate împreună pentru două scopuri distincte:
  - ca un index într-un vector când dimensiunea elementului este diferită de 2, 4 sau 8 octeți – deplasamentul codifică offset-ul față de începutul vector; registrul de bază conține rezultatele unui calcul pentru a determina compensarea cu un anumit element din vecto;
  - pentru a accesa un câmp al unei structuri de date: registrul de bază deține adresa începutului înregistrării, în timp ce deplasamentul constituie offset-ul elementului structurii.

  Un caz special al acestei combinații este accesul la parametrii unei proceduri. Dacă o procedură creează la începutul său un *stack frame* utilizând registrul `RBP`, acest registru este cea mai bună alegere pentru registrul de bază, deoarece selectează automat segmentul de stivă. Această metodă oferă o codificare compactă pentru accesul la parametrii și la variabilele locale ale unei proceduri.
- **Indexată și scalată cu deplasament** – Acest mod de adresare oferă un mod eficient de indexare într-un vector static când dimensiunea elementului este de 2, 4 sau 8 octeți. Registrul de bază conține adresa de început a vectorului, registrul de index conține poziția elementului dorit, iar procesorul convertește automat adresa elementului aplicând factorul de scalare indexului.
- **Bazată, Indexată cu deplasament** – Utilizarea a două registre permite accesarea unui vector bidimensional de octeți (baza conține adresa începutului tabloului), sau un vector unidimensional de structuri de date. (deplasamentul compensează pentru poziția înregistrării în cadrul structurii).
- **Bazată, indexată și scalată cu deplasament** – Utilizarea tuturor componentelor de adresare permite indexarea eficientă a unui vector bidimensional (matrice) atunci când elementele vectorului au dimensiunea de 2, 4 sau 8 octeți.
- **Relativă (`RIP` cu deplasament)** – În modul pe 64 biți, adresarea relativă față de valoarea registrului `RIP` folosește un deplasament cu semn pe 32 de biți pentru a calcula adresa efectivă față de instrucțiunea următoare, prin extinderea semnului valorii pe 32 biți și adăugarea acesteia la valoarea de 64 biți din registrul `RIP`.

2.5. Sintaxa UASM pentru modurile de adresare x86-64

Modurile de adresare a memoriei x86-64 oferă accesul flexibil la memorie, permițând programatorului accesul cu ușurință la variabile, vectori, înregistrări, *pointer*-i și alte tipuri de date complexe. Cunoașterea modurilor de adresare ale procesorului constituie primul pas către înțelegerea limbajului de asamblare 80x86.

Asamblorul `UASM` folosește o sintaxă compatibilă `MASM` pentru a oferi mai multe variante diferite pentru a indica diferitele moduri de adresare: indexate, bazate/indexate cu sau fără deplasament. Asamblorul `UASM` oferă mai multe forme interschimbabile pentru adresarea indirectă a memoriei.

2.5.1. Adresarea imediată

Un operand imediat are o valoare constantă sau o expresie constantă. Când o instrucțiune cu doi operanzi utilizează adresarea imediată, primul operand poate fi un registru sau o locație de memorie, iar al doilea operand este o valoare imediată. Primul operand definește dimensiunea datelor.

```
mov    rax, 110
mov    Var, 100+10
mov    al, '*'
```

2.5.2. Adresarea directă

În modul de adresare directă, adresa efectivă este specificată direct ca parte a instrucțiunii, de obicei indicată de numele variabilei. Asamblorul menține un tabel de simboluri, care stochează adresele tuturor variabilelor utilizate în program.

Adresarea directă se poate face în două forme:

- fie adresa efectivă este specificată ca valoare imediată și precizată *între paranteze pătrate*; de exemplu copierea în registrul `RAX` a valorii de la adresa de memorie `0x000007ff610a9305C` se realizează prin instrucțiunea:

```
mov    rax, [000007ff610a9305Ch]
```

- fie adresa efectivă este specificată prin evaluarea unei expresii constante pe baza simbolul (numele) atașat unei variabile; de exemplu copierea în registrul `RAX` a valorii variabilei cu numele `QwordVar` se realizează prin una din instrucțiunile:

```
mov    rax, QwordVar
mov    rax, [QwordVar]
```

Observație: `UASM` tratează simbolurile „[]” la fel ca operatorul „+”. Acest operator este comutativ, la fel ca operatorul „+” (desigur, aceasta se aplică tuturor modurilor de adresare). Astfel instrucțiunile de mai jos sunt echivalente:

```
mov    rax, [Array+4]      ; RAX = valoarea de la adresa Array+4
mov    rax, Array+4
mov    rax, [Array]+4
mov    rax, Array[4]
mov    rax, 4[Array]
mov    rax, [4][Array]
```

```
mov    rax, [Array][4]
```

2.5.3. Adresarea indirectă

La adresarea indirectă valoarea adresei efective se obține în urma unui calcul pe baza a măcar unui registru de uz general pe 64 biți, conform schemei de calcul din Figura 2.11.

2.5.3.1. Adresarea bazată

În modul de adresare bazată, adresa efectivă este specificată prin valoarea unui registru de uz general pe 64 biți. Ca sintaxă registrul este încadrat între paranteze pătrate. De exemplu dacă considerăm că registrul RBX conține adresa unei locații de memorie pe 16 biți, copierea valorii locației în registrul AX se realizează prin:

```
mov   ax,  [rbx]
```

2.5.3.2. Adresarea bazată cu deplasament

În modul de adresare bazată cu deplasament adresa efectivă este specificată prin valoarea unui registru de uz general pe 64 biți căruia i se adaugă o valoare imediată pe 32 biți în reprezentare cu semn. De exemplu instrucțiunile de mai jos sunt forme echivalente ale aceleiași instrucțiuni:

```
mov    rax, [rbp+16]       ; RAX = valoarea de la adresa RBP+16
mov    rax, [16+rbp]
mov    rax, [16][rbp]
mov    rax, [rbp][16]
mov    rax, 16[rbp]
mov    rax, [rbp]+16
```

2.5.3.3. Adresarea bazată și indexată

În modul de adresare indexată adrese efectivă este specificată prin suma valorii unui registru de uz general pe 64 biți (care constituie baza), și a valorii unui alt registru de uz general pe 64 biți (diferit de RSP) care poate fi multiplicată cu constanta 2, 4 sau 8. În exemplele de mai jos se regăsesc câteva forme echivalente de adresare bazată și indexată:

```
mov    rax, [rbx+rdi*4]
mov    rax, [rdi*4+rbx]
mov    rax, [rdi*4][rbx]
mov    rax, [rbx][rdi*4]
mov    rax, [rbx+4*rdi]
mov    rax, [4*rdi][rbx]          ; etc.
```

2.5.3.4. Adresarea bazată și indexată cu deplasament

În plus față de adresarea bazată și indexată la adresarea bazată și indexată cu deplasament se mai adaugă și o constantă pe 32 biți cu semn care reprezintă un deplasament. Forme echivalente ale aceleiași instrucțiuni sunt:

```
mov    rax, [rbp+rsi*2-24]
```

```
                    mov     rax, [rsi*2+rbp-24]
                    mov     rax, [2*rsi+rbp-24]
                    mov     rax, [-24+2*rsi+rbp]
                    mov     rax, -24[rbp+2*rsi]
                    mov     rax, -24[2*rsi+rbp]
                    mov     rax, -24[rsi*2][rbp]
                    mov     rax, [rsi*2][-24][rbp]
                    mov     rax, [rsi*2][rbp]-24            ; etc.
```

Capitolul 3. Variabile și structuri de date

Variabilele sunt utilizate pentru a stoca informații care trebuie referite și manipulate într-un program. De asemenea, acestea oferă o modalitate de etichetare a datelor cu un nume descriptiv, astfel încât programele să poată fi înțelese mai ușor atât de către alți programatori cât și de autorii acestora mai ales în procesul de depanare și testare. Variabilele pot fi văzute ca niște containere care conțin informații. Singurul lor scop este etichetarea și stocarea datelor în memorie.

Denumirea variabilelor este cunoscută ca una dintre cele mai dificile sarcini ale programatorului. Numele atribuit variabilelor trebuie să fie cât mai descriptiv (dar și suficient de scurt) pentru a fi ușor de înțeles scopul acestora atât de către alți programatori cât și de autorul programului când acesta își recitește program scris cu ceva timp în urmă.

3.1. Declararea variabilelor în limbaj de asamblare

Când computerul accesează o variabilă acesta cumva își „amintește” valoarea acesteia pentru a putea fi folosită ulterior. Pentru a realiza acest lucru, compilatorul alocă o locație de memorie pentru utilizare exclusivă a variabilei respective.

Variabilele care conțin valori unice se numesc variabile *scalare*. Structuri de date care nu sunt scalare includ tablouri (*arrays*), înregistrări (*records*), structuri și liste. Aceste tipuri de date sunt alcătuite din valori scalare și se numesc tipuri de date *compozite*.

3.2. Declararea și accesarea variabilelor scalare

Pentru a declara în limbajul de asamblare o variabilă în segmentul de date, se utilizează o declarație de genul în segmentul de date al programului:

```
.data
        octet  db ?
```

unde `octet` reprezintă o *etichetă*. O etichetă (*label*) într-un limbaj de programare reprezintă o secvență de caractere care identifică o locație (adresă) în cadrul codului sursă. Variabilele globale trebuie declarate în segmentul de date (după declarația `.data`). În asamblare numele etichetelor pot conține litere, cifre, caracterele `'_'`, `'?'` sau `'@'` dar nu pot începe cu o cifră.

Un *segment de date* (declarat cu `.data`) este o porțiune dintr-un fișier obiect sau spațiul de adresă corespunzător al unui program care conține variabile statice inițializate, adică variabile globale și variabile statice locale. Mărimea acestui segment este determinată de dimensiunea variabilelor din codul sursă al programului și nu se poate modifica în timpul rulării programului.

Pentru a declara mai multe variabile într-un program, pur și simplu de adaugă linii suplimentare în segmentul de date (după declarația `.data`) care definește acele variabile, cu observația că numele variabilelor trebuie să fie unice. Asamblorul `UASM` va aloca automat o locație de memorie unică pentru fiecare variabilă. După declararea unei variabile, `UASM` va permite referirea acelei variabilă după nume, mai degrabă decât după locația din segmentul de date. De exemplu, după inserarea declarației de mai sus în segmentul de date, pot fi utilizate în segmentul de cod (după declarația `.code`) instrucțiuni precum

```
        mov    octet, al
```

Prima variabilă declarată în segmentul de date primește chiar adresa de început a segmentului de date. Pentru următoarea variabilă declarată se va aloca adresa imediat următoare după variabila anterioară. De exemplu, dacă variabila de la locația zero a fost o variabilă de un

octet, următoarea variabilă primește stocare alocată la adresa cu un deplasament (*offset*) `+1` față de adresa de început. Dacă de exemplu prima variabilă a fost un cuvânt (`word`), a doua variabilă primește locația de la adresa segmentului de date cu un deplasament `+2`. Asamblorul este întotdeauna atent să aloce variabile în așa fel încât acestea să nu se suprapună. Fie următoarea secvență de declarații într-un segment de date:

```
.data
        octet        byte  ?      ; byte aloca octeti
        cuvant       word  ?      ; word aloca cuvinte
        dublu_cuvant dword ?      ; dword aloca dublu-cuvinte
        cuvant2      byte  ?
        octet2       word  ?
```

Asamblorul `UASM` va aloca pentru variabila `octet` adresa cu un offset `+0`. Deoarece `octet` are dimensiunea de un octet, următoarea locație de memorie disponibilă va fi cea cu offset de `+1`. Prin urmare, asamblorul va aloca spațiu de stocare pentru variabila `cuvant` la locația cu offset de `+1`. Deoarece cuvintele necesită doi octeți, următoarea locație de memorie disponibilă după `cuvant` are un deplasament de `+3`, unde asamblorul va aloca spațiu de stocare pentru `dublu_cuvant`. Variabila `dublu_cuvant` are dimensiunea de patru octeți, astfel încât asamblorul va aloca adresa cu deplasament de `+7` pentru `cuvant2`, și pentru `octet2` adresa cu deplasament de `+9`. Dacă s-ar adăuga la declarațiile de mai sus încă o variabilă, asamblorul ar aloca spațiu de stocare pentru acesta la locația cu deplasament `+10` (sau `+0Ah` în hexazecimal).

Ori de câte ori se va referi ulterior în program una dintre variabilele declarate în segmentul de date, asamblorul va înlocui automat numele cu adresa corespunzătoare. Astfel, asamblorul permite utilizarea de nume simbolice pentru variabilele unui program și se poate ignora complet faptul că aceste variabile sunt de fapt locații de memorie în segmentul de date.

3.2.1. Declararea și utilizarea variabilelor BYTE

Orice tip de date care are mai puțin de 256 de valori diferite se poate reprezenta cu un singur octet. Aceasta include câteva tipuri de date foarte importante și deseori utilizate, inclusiv tipul de date caracter, tipul de date boolean, enumerările și tipurile de date întregi mici (cu și fără semn).

Caracterele dintr-un sistem de calcul compatibil IBM utilizează setul de caractere ASCII reprezentat pe opt biți. De aceea, se poate constata că majoritatea variabilelor de tip `byte` dintr-un program tipic conțin date de tip caracter, în special șiruri de caractere.

Tipul de date *boolean* reprezintă doar două valori: adevărat sau fals. Prin urmare, este nevoie de un singur bit pentru a reprezenta o valoare booleană. Cu toate acestea, procesoarele x86 pot lucra doar cu date de multiplu opt biți lățime. De fapt, este nevoie de instrucțiuni suplimentare pentru a manipula un singur bit, mai degrabă decât un octet întreg. Prin urmare, este mai eficientă utilizarea unui octet pentru a reprezenta o valoare booleană. Majoritatea programatorilor folosesc valoarea `0` (zero) pentru a reprezenta „fals” și orice altceva (de obicei `1` sau `-1`) pentru a reprezenta valoarea de „adevărat”. Flag-ul de zero (`ZF`) al procesoarelor x86 face foarte ușor testarea zero/nu zero. Trebuie reținut faptul că această alegere de zero sau non-zero este în principal pentru comoditate, se pot utiliza oricare alte două valori diferite pentru a reprezenta „adevărat” și „fals”.

Majoritatea limbajelor de nivel înalt care acceptă tipuri de date enumerări, le convertesc (intern) în numere întregi fără semn. Primul element al listei este, în general, elementul zero, al doilea element din listă este elementul unu, al treilea este elementul doi, etc.

Desigur, dacă este necesară reprezentarea unei valori întregi fără semn în intervalul

`0..255` sau un număr întreg cu semn `-128..127`, o variabilă de singur octet este cea mai bună variantă în cele mai multe cazuri. Majoritatea programatorilor tratează toate tipurile de date, cu excepția numerelor întregi cu semn mici, ca valori fără semn. Adică, caracterele, valorile logice (*boolean*), enumerările și numerele întregi fără semn sunt de obicei valori fără semn. În unele cazuri foarte speciale se tratează un caracter ca o valoare fără semn, dar de cele mai multe ori chiar și caracterele sunt valori fără semn.

Există trei declarații principale pentru definirea variabilelor de octeți într-un program. Acestea sunt:

```
            identificator db ?
            identificator byte ?
            identificator sbyte ?
```

unde `identificator` reprezintă numele variabilei de octet. Directiva „`db`" este o formă mai veche, anterioară versiunii `MASM` 6.x (cu care `UASM` este compatibil). Deși această directivă este folosită destul de mult de către programatori, Microsoft consideră că este un termen învechit și recomandă utilizarea declarațiilor `byte` și `sbyte`.

Directiva „`byte`" declară variabile de tip octet fără semn. Această declarație se utilizează pentru toate variabilele de tip octet, cu excepția numerelor întregi mici cu semn. Pentru valori întregi cu semn se poate utilizeaza directiva „`sbyte`" (engl. *signed byte*).

Odată declarate variabilele de tip octet, acestea pot fi referite în cadrul programelor prin intermediul numelui lor:

```
  .data
            x db ?
            y byte ?
            t sbyte ?
  .code
; ...
            mov x, 0
            mov y, 129
            mov t, -50
            mov al, x
            mov y, cl
; ...
```

Asamblorul efectuează o verificare a tipului datelor doar într-o mică măsură, de fapt, acesta va verifica valorile vehiculate doar pentru a identifica dacă acestea „încap" (se pot reprezenta) în locația țintă. Următoarele instrucțiuni cu variabilele declarate anterior sunt legale în `UASM`:

```
            mov x, -127
            mov y, -1
            mov t, 253
```

Deoarece toate aceste variabile sunt variabile de dimensiune un octet și toate constantele asociate se vor încadra în opt biți, asamblorul va permite aceste operații. Cu toate acestea, ele sunt logic incorecte. Ce înseamnă încărcarea valorii `-1` într-o variabilă pe octet fără semn? Deoarece valorile unui număr cu semn pe opt biți trebuie să fie în intervalul `-128..127`, ce se întâmplă atunci când valoarea `253` este stocată într-o variabilă de tip octet cu semn? Ei bine, asamblorul convertește pur și simplu aceste valori în echivalenții lor pe opt biți (`-1` devine `0FFh=255`, `253` devine `0FDh=-3` etc.).

Deși asamblorul nu verifică dacă ambii operanzi ai unei instrucțiuni sunt cu sau fără semn, cu siguranță verifică dimensiunea acestora. Dacă dimensiunile lor nu sunt egale, asamblorul va genera un mesaj de eroare adecvat. Următoarele exemple sunt ilegale:

```
mov    x, ax        ; nu se poate copia o val. pe 16 bits in una pe 8.
mov    y, 300       ; 300 nu se poate reprezenta pe 8 biti.
mov    t, -130      ; -130 nu se poate reprezenta pe 8 biti.
```

Faptul că ansamblul nu face cu adevărat diferența între valorile cu semn și cele fără semn nu înseamnă că directivele `byte` și `sbyte` sunt inutile. Există două avantaje: în primul rând crește lizibilitatea programelor, iar în al doilea rând faptul că instrucțiunea `mov` ignoră diferența, nu înseamnă că și alte instrucțiuni o fac.

Un fapt demn de menționat cu privire la declararea variabilelor de tip octet în toate declarațiile anterioare este prezența caracterului `'?'` în declarația variabilelor. Acest semn de întrebare specifică faptul că variabila va fi lăsată neinițializată atunci când programul este încărcat în memorie. Se poate specifica o valoare inițială pentru variabile, care va fi încărcată în memorie înainte ca programul să intre în execuție, prin simpla înlocuire a semnul întrebării cu valoarea inițială (ca în exemplul de mai jos).

```
x db 0
y octet 250
t sbyte -100
```

În acest exemplu, asamblorul va inițializa variabilele `x`, `y` și `t` cu `0`, `250` respectiv, `-100`, atunci când programul se încarcă în memorie. Acest fapt se va dovedi util, mai ales în cazul tabelelor și a vectorilor. Încă o dată, asamblorul verifică doar dimensiunile valorilor și nu va verifica valoarea pentru un `byte` este pozitivă sau dacă valoarea unui `sbyte` este în intervalul `-128..127`. Asamblorul va permite, astfel, orice valoare în intervalul `-128..255` atât pentru `byte` cât și pentru `sbyte`.

3.2.2. Declararea și utilizarea variabilelor WORD și DWORD

Majoritatea programelor folosesc valori de tip `word` și `dword` pentru reprezentarea numerelor întregi cu sau fără semn pe 16 respectiv 32 biți. Desigur, orice variabilă de tip `word` poate lua o valoare în intervalul `-32768..65535` (uniunea intervalului pentru constantele cu și fără semn pe 16 biți), iar variabilele de tip `dword` pot lua valori în intervalul `-2147483648..4294967295` (uniunea intervalului pentru constantele cu și fără semn pe 32 biți).

Pentru a declara variabile pe 16 biți (doi octeți) se utilizează directivele `dw`, `word` și `sword`. Următoarele exemple demonstrează utilizarea lor:

```
_int16      dw ?
uint16      word ?
int16       sword ?
_Int16      word 0
uInt16      sword -5
Int16       dw 50000
```

Pentru variabile pe 32 biți (patru octeți) se utilizează directivele `dd`, `dword` și `sdword`, cum se poate observa în exemplele:

```
_int32      dd ?
uint32      dword ?
int32       sdword ?
```

```
_Int32      dword 0FFFFFFFFh
uInt32      sdword -1000000000
Int32       dd 3000000000
```

Încă o dată, merită subliniat faptul că asamblorul nu verifică tipurile acestor variabile atunci când sunt introduse valorile de inițializare. Dacă valoarea se încadrează în 16 biți pentru variabile de tip `word`, respectiv pe 32 de biți pentru `dword`, asamblorul o va accepta. Cu toate acestea, dimensiunea valorilor este strict verificată.

Variabilele de tip `dword` mai pot fi utilizate pentru a stoca numere în virgulă mobilă simplă precizie (32 biți), dar asamblorul `UASM` oferă directive specializate pentru astfel de variabile.

3.2.3. Declararea și utilizarea variabilelor QWORD

Majoritatea programelor x86-64 folosesc valori pe 64 biți (8 octeți) pentru stocarea: numerelor întregi cu semn pe 64 biți, numere întregi fără semn pe 64 biți și adrese în memorie (pointeri).

Declarațiile `dq`, `qword` și `sqword` sunt disponibile pentru a declara variabile pe 64 biți. Exemple de mai jos demonstrează utilizarea lor:

```
_int64      dq ?
uint64      qword ?
int64       sqword ?
_Int64      qword 0161616161616161616h
uInt64      sqword -1
Int64       dq 3000000000
Ptr64       dq uint64
```

Scopul inițial al directive `dq`/`qword` a fost acela de a permite declararea variabilelor în virgulă mobilă cu dublă precizie pe 64 biți și variabile întregi pe 64 biți. Ulterior au fost introduse directive specializate pentru definirea de variabile în virgulă mobilă.

Majoritatea acestor declarații sunt ușoare modificări ale declarațiilor pentru tipurile prezentate anterior. Desigur, puteți inițializa orice variabilă pe 64 biți cu o valoare în intervalul `-9223372036854775808..18446744073709551615` (unirea intervalului pentru numere cu și fără semn pe 64 biți). Ultima declarație de mai sus este însă diferită. În acest caz, apare o etichetă în locul valorii inițiale (mai exact numele variabilei `uint64`). Când apare o etichetă ca valoare de inițializare, asamblorul va înlocui eticheta cu adresa acesteia în memorie. Această formă de inițializare este utilă pentru inițializarea *pointerilor*.

3.2.4. Declararea și utilizarea variabilelor FWORD și TWORD

Asamblorul `UASM` permite, de asemenea, declararea de variabile cu șase și zece octeți folosind directivele `df`/`fword` și `dt`/`tbyte`. Declarațiile care utilizează aceste declarații au fost inițial destinate valorilor în virgulă mobilă și BCD. Există directive specializate pentru variabilele în virgulă mobilă și tipurile de date declarate de aceste directive sunt prezentate pentru completitudine.

Declarația `df`/`fword` a fost introdusă pentru a declara pointeri pe 48 de biți pentru modul protejat pe 32 de biți pe 386 și versiunile ulterioare. Deși această directivă poate fi utilizată pentru a crea o variabilă arbitrară de șase octeți, există directive mai bune pentru a face acest lucru.

Directivele `dt`/`tbyte` declară variabile de zece octeți în spațiul de memorie. Există două

tipuri de date care utilizează tipul de date de zece octeți și sunt specifice familiei 80x87 (co-procesorul matematic): valori BCD de zece octeți și valori în virgulă mobilă de precizie extinsă (80 biți). Tipul de date BCD nu mai este folosit pe mașinile x86-64, iar în ceea ce privește tipul de date în virgulă mobilă, există o modalitate mai bună de a declara aceste variabile.

3.2.5. Declararea variabilelor în virgulă mobilă cu REAL4, REAL8 și REAL10

Acestea sunt directivele care ar trebui utilizate pentru declararea de variabile în virgulă mobilă. La fel ca `dd`, `dq` și `dt`, aceste instrucțiuni rezervă patru, opt și zece octeți. Câmpurile de inițializare pentru aceste declarații pot conține un semn de întrebare (dacă nu doriți să inițializați variabila) sau poate conține o valoare inițială în format în virgulă mobilă. Următoarele exemple demonstrează utilizarea acestora:

```
x       real4 3.14
y       real8 1.0e-12
z       real10 -6.99e+5
```

Câmpul de inițializare trebuie să conțină fie caracterul `'?'` fie o constantă validă în virgulă mobilă utilizând fie notație zecimală, fie științifică. Utilizarea constantelor întregi pentru inițializarea tipurilor de date în virgulă mobilă *nu sunt permise* în `MASM`, care ar fi generat eroare, în schimb asamblorul `UASM` le acceptă și convertește declarațiile de numerele la numere în virgulă mobilă (pentru tipurile de date `real4`, `real8` și `real10`) precum declarația:

```
nr real8 1
```

Asamblorul `UASM` permite, de asemenea, utilizarea directivelor `dd`/`dword`, `dq`/`qword` și `dt`/`tword` pentru a declara variabilelor în virgulă mobilă (deoarece aceste directive rezervă necesarul de patru, opt sau zece octeți de memorie). Se pot inițializa astfel de variabile cu constante în virgulă mobilă în câmpul operand, dar există un dezavantaj major în declararea variabilelor în acest fel: utilizarea directivelor `dd`, `dq` și `dt` permit inițializarea atât cu constante întregi, cât și cu cele în virgulă mobilă dar nu vor promova valorile întregi la numere în virgulă mobilă (spre deosebire de directivele `real4`, `real8` și `real10`). Aceasta ar putea părea o caracteristică bună la prima vedere, dar reprezentarea întreagă pentru valoarea „`1`” nu este aceeași cu reprezentarea în virgulă mobilă pentru valoarea „`1.0`”. Deci, se va inițializa din greșeală cu valoarea întreagă „`1`” în locul valorii corecte de „`1.0`”, asamblorul nu va genera nici o eroare dar programul va oferi rezultate incorecte. Prin urmare, se recomandă întotdeauna utilizarea directivelor `real4`, `real8` și `real10` pentru a declara variabile în virgulă mobilă precizie simplă, dublă respectiv extinsă.

3.2.6. Crearea propriilor nume de tipuri de date cu TYPEDEF

Directiva typedef poate fi utilizată dacă pur și simplu în locul directivelor alese de Microsoft pentru declararea de octeți (`byte`), cuvinte (`word`), dublu-cuvinte (`dword`), numere reale și alte variabile și, spre exemplu, se preferă convenția de numire din C. De exemplu se dorește utilizarea de declarații de tip `__int64` pentru numere întregi cu semn pe 64 de biți, `double` pentru numere în virgula mobilă pe 64 biți, `char` pentru caractere sau orice altceva. În C se poate folosi directiva `#define` sau o declarație `typedef` pentru a îndeplini această sarcina. Asamblorul are propria declarație `typedef` care permite, de asemenea, crearea de aliasuri ale acestor nume. Următorul exemplu arată modalitatea în care putem utiliza aceste declarații în asamblare:

```
__int64      typedef sqword
char         typedef byte
double       typedef real8
```

După aceste declarații se pot declara variabilele cu declarații mai sugestive precum:

```
chr    char '$'
nr     double 1.0
int64  __int64 ?
```

Singurul dezavantaj al utilizării tipurilor proprii de date este acela că în cazul utilizării unui editor care suportă „*sintax highlighting*” pentru asamblare, tipurile de date proprii nu vor fi recunoscute ca fiind cuvinte rezervate și nu vor mai fi colorate cu culoarea specifică directivelor.

3.2.7. Declararea și utilizarea pointer-ilor

În asamblarea pentru x86-64 pointerii sunt date scalare reprezentate pe 64 biți (`QWORD`). Cei mai mulți programatori au contact cu pointerii în limbaje de programare de nivel înalt precum C și marea majoritatea dintre aceștia au o experiență neplăcută atunci când întâlnesc pentru prima dată *pointer*-ii într-un limbaj de nivel înalt. Pointerii sunt de fapt mai ușor de tratat în limbajul de asamblare. În plus, majoritatea problemelor legate de *pointer*-i probabil nu au nimic de-a face cu *pointer*-ii, ci mai degrabă cu structurile de date pe care aceștia încercau să le implementeze. *Pointer*-ii, pe de altă parte, au o mulțime de utilizări în limbajul de asamblare care nu au nimic de-a face cu listele înlănțuite, arbori și alte structuri de date. Structurile de date simple, cum ar fi tablourile și vectorii, implică adesea utilizarea *pointer*-ilor.

Din păcate, limbajele de nivel înalt, cum ar fi C, tind să ascundă simplitatea utilizării *pointer*-ilor în spatele unui perete de abstractizare. Această complexitate suplimentară (care există, din motive întemeiate, apropo) tinde să-i sperie pe unii programatori, deoarece nu înțeleg ce reprezintă de fapt *pointer*-ii.

Să considerăm, ca un simplu exemplu, un vector de 4096 numere fără semn pe 64 de biți definit în C de declarația:

```
unsigned __int16 m[4096];
```

Conceptul destul de ușor de înțeles: „`m`” este un vector (șir) cu „`4096`” numere întregi fără semn pe 64 biți, cu indecși de la `m[0]` la `m[4095]`. Fiecare dintre aceste elemente acestui vector poate stoca o valoare întreagă fără semn pe 64 biți care este independentă de celelalte elemente. Cu alte cuvinte, acest vector conține 4096 variabile diferite, la care se face referire printr-un număr (indicele vectorului), mai degrabă decât după nume.

Dacă un program conține declarația `m[0] = 10`, aceasta înseamnă că valoarea `10` va fi stocată în primul element al vectorului. Fie următoarea secvență C:

```
unsigned __int64 = 0;
m[0] = 10;
```

în care se observă că cele două operații efectuează exact aceeași operație ca și `m[0] = 10` și deci se poate trage concluzia că orice variabilă „întreagă” cu valori între 0 și 4095 poate fi utilizată ca index pentru vectorul `m[]`.

Acum dacă se analizează secvența:

```
m[1] = 0;
m[ m[1] ] = 10;
```

se poate observa că aceste două instrucțiuni efectuează exact aceeași operație. Prima declarație

stochează valoarea „0” în elementul vectorului m[1]. A doua instrucțiune preia valoarea lui m[1], care este un număr „întreg”, astfel încât poate fi folosit ca un index în vectorul m[] și folosește acea valoare („0”) pentru a controla unde se stochează valoarea 10.

Dacă secvența de exemplele de mai sus par rezonabile, poate bizare, dar cu toate acestea utilizabile, atunci *pointer*-ii sunt ușor de înțeles. Aceasta pentru că m[1] este un *pointer*! Ei bine, nu chiar, dar dacă se mărește dimensiune vectorului la „m” la dimensiunea memoriei sistemului și se tratează acest vector ca fiin toată memoria sistemului, se obține definiția exactă a unui *pointer*: un *pointer* este pur și simplu o locație de memorie a cărei valoare reprezintă adresa altei locații de memorie. *Pointer*-ii sunt foarte ușor de declarat și de utilizat într-un program în limbaj de asamblare.

Un pointer este o valoare pe 64 biți care reprezintă o adresă în memorie. Dacă aveți o variabilă p de tip qword care conține o adresă la o locația de memorie de tip byte de exemplu, se poate utiliza secventa următoare:

```
mov rbx, p                ; se incarca rbx cu valoarea "pointer"-ului
mov al, byte ptr [rbx]    ; se acceseaza data "pointata" de var. P
```

În asamblare nu se poate accesa direct o locație stocată într-o variabilă, datorită limitării arhitecturii sistemelor x86, și anume faptul că într-o singură instrucțiune nu se poate accesa decât o singură locație de memorie. Un acces indirect printr-o variabilă ar însemna accesarea atât a valorii variabilei cât și a locației de la adresa stocată în acea variabilă, de aceea accesarea indirectă se face întotdeauna prin intermediul regiștrilor de uz general pe 64 biți. Pentru a specifica faptul că se accesează locația de la adresa din registru și nu valoarea acestuia, numele registrului se încadrează în paranteze pătrate (a se vedea secțiunea 2.5.3. Adresarea indirectă).

Pentru a încărca într-un registru adresa unei variabile se utilizează instrucțiunea LEA (a se vedea secțiunea 4.2.4. Instrucțiunea LEA).

3.3. Declararea și accesare tipurilor de date compozite

Tipurile de date compozite sunt cele care sunt construite din alte tipuri de date (în general scalare). Un vector este un bun exemplu de tip de date compozit – este un agregat de elemente de același tip. Un tip de date compozit nu trebuie să fie compus din tipuri de date scalare, pot exista matrici de matrici, de exemplu, dar în cele din urmă acestea se descompun în tipuri primare, scalare.

3.3.1. Tablouri

Tablourile (*arrays*) sunt probabil cel mai frecvent utilizat tip de date compozit. Cu toate acestea, majoritatea programatorilor începători au o înțelegere deficitară a modului în care funcționează tablourile și a compromisurilor de eficiență asociate utilizării acestora. Este surprinzător câți dintre programatorii începători (și nu numai) ajung să vadă tablourile într-o perspectivă complet diferită odată ce descoperă cum funcționează acestea la nivel mașină.

În mod abstract, un tablou (*array*) este un tip de date compozit ai cărui membri (elemente) sunt toate de același tip. Selectarea unui element din tablou se face printr-un index întreg. Indecși diferiți selectează elemente unice ale tabloului, iar în cele mai multe cazuri indecșii întregi sunt adiacenți (deși nu este obligatoriu), adică elementele tabloului ocupă locații continui în memorie.

Adresa de bază a unui tabel este adresa primului element din matrice și apare întotdeauna în cea mai mică locație de memorie (de obicei cea cu indexul 0). Al doilea element al tabelului îl urmează direct pe primul din memorie, al treilea element îl urmează pe cel de-al doilea, etc.. Nu

exista nici o regulă care să impună ca indecșii să înceapă de la zero, teoretic indecșii pot începe cu orice număr atât timp cât sunt continui. Cu toate acestea, pentru comoditate, accesarea elementelor unui tabel este mai facilă dacă primul index este zero. În general, în majoritatea cazurilor primul element se află la indexul zero, cu excepția cazului în care există un motiv întemeiat pentru a începe de la alt index. Cu toate acestea, acest lucru este doar pentru consecvență, nu există niciun beneficiu de eficiență într-un fel sau altul pentru a porni indexul tabelului de la o altă valoare decât zero.

Pentru a accesa un element al unui tablou, aveți nevoie de o funcție care să convertească indexul tabelului în adresa fizică a elementului indexat. Pentru un tablou cu o singură dimensiune, cu index inițial zero, această funcție este foarte simplă:

```
Adresa_element = Adresa_bază + (Index * Dimensiune_element)
```

unde `Dimensiune_element` este dimensiunea în octeți a unui element al tabelului.

3.3.1.1. Declararea tablourilor unidimensionale (vectori sau șiruri)

Declarațiile tablourilor unidimensionale se bazează pe declarațiile prezentate anterior. Pentru a aloca `n` elemente într-un tablou unidimensional (vector sau șir), se utilizează în segmentul de date o declarație de forma:

```
nume_vector tip_de_baza n dup (?)
```

`Nume_vector` este numele variabilei tabelului iar `tip_de_baza` este tipul unui element al tabelului respectiv. Această declarație alocă spațiul de stocare (memorie) pentru tablou. Pentru a obține adresa de bază a tabloului, se utilizează pur și simplu doar `nume_vector`, deoarece acesta este de fapt o etichetă care reprezintă adresa (locația) curentă în spațiul de stocare (memorie).

Directiva `DUP` este utilizată pentru a multiplica (engl. *duplicate*) de numărul de ori ce precede directiva obiectul din paranteze. Astfel deoarece în construcția „`n dup (?)`” apare caracterul `'?'`, definiția de mai va crea un vector de `n` valori neinițializate. Fie următoarele exemple de vectori neinițializate:

```
char_vect  byte 256 dup (?)          ; vect[0..255] of byte
PTR_vect   qword 10 dup(?)           ; vect[0..9] of qword/pointer
float_vect real4 1024 dup(?)         ; vect[0..1023] of real4/float
```

Elementele unui vector se pot inițializa cu o anumită valoare utilizând declarații de forma:

```
char_array  byte 10 dup ('*')
PTR_vect    qword 10 dup(0)
float_vect  real4 10 dup (-1.0)
```

Ambele definiții creează vectori cu zece elemente. Prima definiție inițializează fiecare caracter sin șirul de caractere `char_array` cu valoarea „`42`” (cod ASCII pentru caracterul `'*'`), a doua declarație inițializează elementele vectorului de pointeri cu valoarea „`0`” (*pointer*-ul `NULL`), iar ultima declarație inițializează fiecare element al vectorului de numere în virgulă mobilă simplă precizie cu valoarea „`-1.0`”.

Acest mecanism de inițializare se utilizează doar în cazul în care toate elementele trebuie inițializate cu aceeași valoare. Dacă fiecare element al tabloului trebuie inițializat cu o valoare diferită declarația pentru un astfel de vector va avea forma:

```
nume_vector tip_de_baza val₁, val₂, val₃, ..., valₙ
```

Acest format alocă `n` variabile de tipul `tip_de_baza`. Acesta inițializează primul element

cu valoarea val_1, al doilea element cu valoarea val_2, etc.. Deci, prin simpla enumerare a fiecărei valori se poate crea un vector cu valorile inițiale dorite. Astfel un vector al primelor zece nume prime va arăta astfel:

```
prim word 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

Dacă vectorul are mai multe elemente și se dorește scrierea acestora pe linii diferite, există mai multe moduri de a continua declarația pe linia următoare. Cea mai simplă metodă este utilizarea unei alte declarații, dar fără etichetă:

```
prim word 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
     word 31, 37, 41, 43, 47, 53, 59, 61, 67
     word 71, 73, 79, 83, 89, 97
```

O altă variantă este utilizarea caracterului `'\'` (*backslash*) la finalul fiecărei linii pentru a specifica asamblorului să continue citirea datelor pe următoarea linie:

```
prim word 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 \
          31, 37, 41, 43, 47, 53, 59, 61, 67 \
          71, 73, 79, 83, 89, 97
```

Pentru inițializarea matricelor unidimensionale (vectori sau șiruri) directiva `dup` nu este folosită doar pentru multiplicarea unui singur element al vectorului. Directiva `dup` multiplică orice construcție (specificată în paranteze), precum în următoarea declarație:

```
sir byte 10 dup ('a', 'b', 'c', 'd', 'e')
```

Acest sir de caractere (vector de octeți) are 50 de elemente și nu 10. În declarația de mai sus asamblorul va multiplica conținutul dintre paranteze de 10 de ori, și deoarece există cinci elemente în paranteze, operatorul `dup` creează `10*5` sau `50` elemente ale șirului de caractere. Valoarea inițială a șirului va fi: `"abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcde"` (observație: șirul declarat anterioară nu va conține terminatorul de șir `0` sau `'\0'`).

3.3.1.2. Accesarea elementelor unui tablou unidimensional

Pentru a accesa un element al unui tablou unidimensional cu indexul începând de la valoarea zero, se utilizează formula prezentata anterior:

```
Adresa_element = Adresa_bază + (Index * Dimensiune_element)
```

Adresa de bază (valoarea `Adresa_bază`) a unui vector se obține pe baza numelui vectorului (deoarece asamblorul asociază adresa primului element cu eticheta care reprezintă numele variabilei). Valoarea pentru `Dimensiune_element` este dată de numărul de octeți pentru fiecare element de matrice. (de exemplu dacă obiectul este un vector de octeți, câmpul `Dimensiune_element` are valoarea `1`). Pentru vectorul „`prim`” definit anterior care are valori de tip `word` (2 octeți) formula de calcul a adresei unui element este

```
Adresa_element = prim + (index * 2)
```

O secvență de cod pentru a încărca în registrul `AX` valoarea „`index`” din vectorul „`prim`” este (echivalentă operației `AX=prim[index]`):

```
lea rbx, prim              ; RBX = adresa variabilei prim
add rbx, index             ; RBX = prim + Index
add rbx, index             ; RBX = prim + Index * 2
mov ax, [rbx]              ; AX = [RBX] locația de la adresa din RBX
```

Arhitectura x86-64 oferă o modalitate nativă pentru accesarea elementelor dintr-un vector

unidimensional de elemente de bază (octeți, cuvinte dublu și cvadruplu-cuvinte) prin intermediul modurilor de adresare. Într-adevăr, însăși sintaxa modurilor de adresare sugerează chiar un acces la un tablou unidimensional. În cazul în care elementele vectorului nu sunt date de dimensiune `1`, `2`, `4` sau `8` indexul trebuie înmulțit cu dimensiunea elementului înainte de utilizarea modului de adresare. Secvența utilizând modurile de adresare ale procesorului x86-64 pentru implementarea operației `AX=prim[index]` este:

```
lea rbx, prim              ; RBX = adresa variabilei prim
mov rsi, index             ; RSI = index
mov ax, [rbx+rsi*2]        ; AX = prim[Index] // 2 = sizeof(word)
```

Față de prim a implementare în care secvența de instrucțiuni calculează în mod explicit suma adresei de bază plus indexul multiplicat de două ori, cea de-a doua implementare se bazează pe modul de adresare bazat și indexat pentru a calcula implicit valoarea adresei. În această din urmă implementare registrul `RBX` conține adresa de bază, iar registrul `RSI` conține valoarea indexului. Faptul ca valoarea adresei de bază se păstrează în registrul `RBX` oferă un avantaj în cazul accesării mai multor locații dintr-un vector prin faptul că registrul `RBX` nu mai trebuie (re)încărcat cu adresa de bază a vectorului pentru următorul acces. Desigur, cu cât vectorul este accesat mai des fără a reîncărca registrul cu valoarea de bază, secvența de cod va fi mai rapidă. Astfel în secvențe de cod complexe, se poate obține un spor de performantă semnificativ.

3.3.1.3. Tablouri multidimensionale

Hardware-ul x86-64 poate gestiona cu ușurință tablouri unidimensionale prin intermediul modurilor de adresare. Din păcate, nu există un mod de adresare care să vă permită accesarea cu ușurință a elementelor tablourilor multidimensionale (matrici). Prima problemă constă în modalitatea în care un obiect multidimensional este stocat într-un spațiu de memorie unidimensional, și asta deoarece memoria sistemelor de calcul x86-64 este, de fapt, un „vector” unidimensional accesat prin intermediul unui „index” care este adresa pe 64 biți a locației de memorie.

Fie o declarație tip C pentru o matrice de 4x4 numere în virgula mobilă simplă precizie:

```
float M[4][4];
```

Această matrice conține 16 elemente organizate ca patru rânduri de câte patru numere. Cumva trebuie realizată o corespondență între cele 16 numere ale matricii și cei 64 octeți adiacenți în memoria principală (un număr în simplă precizie ocupă 4 octeți consecutivi în memorie).

Maparea efectivă nu este importantă atât timp cât îndeplinite două condiții:

- fiecare element se mapează la o locație de memorie unică (adică nu există două elemente ale matricii care ocupă aceleași locații de memorie);
- maparea este consecventă (adică, unui anumit element din matrice îi corespunde întotdeauna aceleași locații de memorie)

Deci, va fi nevoie de o funcție bijectivă cu doi parametri de intrare (rând și coloană) care produce un deplasament (engl. *offset*) într-un vector liniar de 64 octeți (16 numere pe câte 4 octeți). Această funcție realizează o operație cunoscută și sub denumirea de liniarizare sau vectorizare a unei matrici. Deși poate fi folosită orice funcție care îndeplinește constrângerile de mai sus, se alege o mapare care este eficientă din punct de vedere al implementării calculului în timpul rulării și funcționează pentru orice matrice multidimensională (și nu doar pentru cea exemplificată: 4x4 bidimensională).

Deși există un număr mare de funcții posibile care pot fi utilizate, majoritatea programatorilor și a limbajelor de nivel înalt folosesc două dintre ele: ordonarea *row-major* și *column-major*. Termenii *row-major* și *column-major* provin din terminologia legată de ordonarea obiectelor. O modalitate generală de a ordona obiecte cu multe atribute este de a grupa și ordona mai întâi după un atribut, apoi, în cadrul unui grup astfel obținut, obiectele să fie grupate și ordonate după un alt atribut etc.. Dacă gruparea/ordonarea se face după mai multe atribute, primul ar fi să fie numit *major* și ultimul *minor*. Dacă doar două atribute participă la ordonare, este suficient să se numească numai atributul *major*. În cazul matricilor, atributele sunt indicii de-a lungul fiecărei dimensiuni. Pentru matrici bidimensionale în *notație matematică*[1], primul index indică rândul, iar al doilea indică coloana. Această convenție este transferată la sintaxa limbajelor de programare de nivel înalt, deși adesea cu indecși care încep de la `0`.

- ***Ordonarea matricilor după linie (row-major)***

Ordonarea matricilor după rânduri (linii) atribuie elemente succesive, parcurgând matricea întâi pe rânduri și apoi în jos pe coloane, către locații de memorie succesive.

Ordonarea *row-major* este metoda utilizată de majoritatea limbajelor de programare de nivel înalt (inclusiv C/C++). Această ordonare este foarte ușor de implementat și ușor de utilizat în limbajul mașinii (în special într-un *debugger*). Conversia dintr-o structură bidimensională într-o matrice liniară este intuitivă: în memorie se va regăsi primul rând (rândul zero) urmat de cel de-al doilea rând până la sfârșitul acestuia, urmat de al treilea rând, ș.a.m.d..

Fie o matrice bidimensională *A* de dimensiune *M*×*N*. Matricea va ocupa astfel în memorie *M***N* locații de dimensiunea elementului de bază al matricii și opțional încă două elemente care constituie numărul de linii și de coloane ale matricii (*M* și *N*). Este uneori utilă stocarea dimensiunilor matricei pentru o eventuala operație inversă, de transformare din vector în matrice sau pentru operațiile de adunare, scădere, transpunere și înmulțire cu matrici liniarizate.

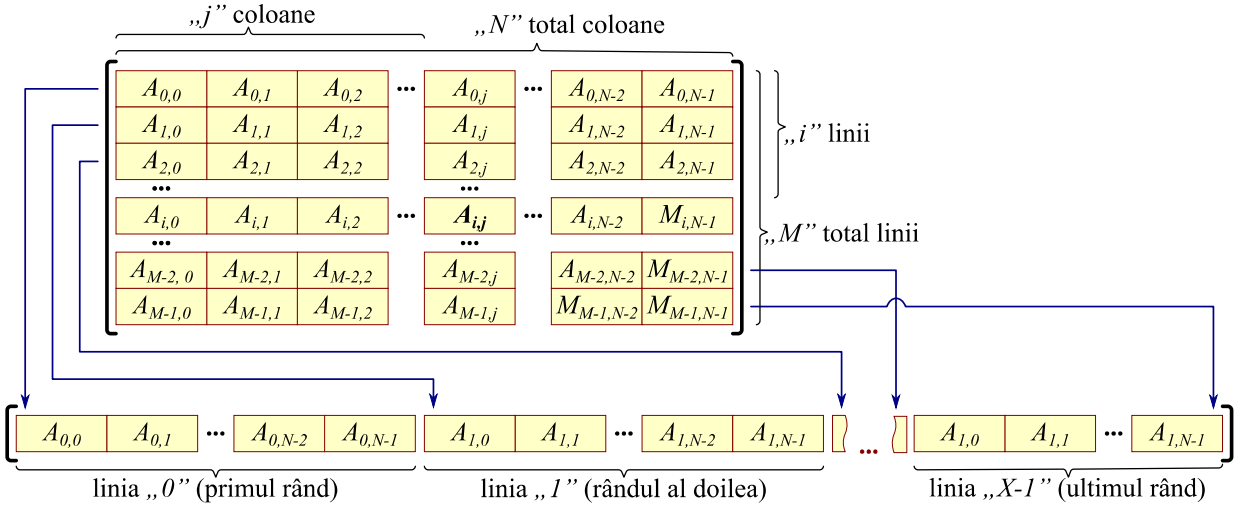

Figura 3.1. Liniarizarea unei matrici.

Adresa elementului *Ai,j* (sintaxa C: `A[i][j]`) aflat de pe linia *i* și coloana *j* a matricei *A* de dimensiune *M*×*N* în memoria calculatorului este dată de formula:

```
addr(A[i][j]) = addr(A[0][0]) + (i*N + j)*sizeof(element)
              = A + (i*N + j)*sizeof(element)
```

În formula anterioară adresa primului element al matricii (notat `A[0][0]`) este de fapt

1 *Notația matematică* reprezintă un sistem de reprezentări simbolice ale obiectelor și ideilor matematice. Aceasta folosește convențional simboluri sau expresii simbolice care sunt destinate să aibă un sens semantic precis.

eticheta zonei de memorie (numele) alocat variabilei (notată `A` și recunoscuta ca atare de către asamblor). Formula se poate deduce foarte ușor: indexul elementului matricii $A_{i,j}$ (notat `A[i][j]`) în forma liniarizată se obține prin "parcurgerea" a „*i*" linii complete (liniile și coloanele fiind indexate de la zero), fiecare având *M* elemente (de aici `i*N`), iar pe linia curentă are „*j*" elemente, relativ la primul element al matricii $A_{0,0}$ (notat `A[0][0]`). Acest index în matricea liniarizată reprezintă numărul de ordine al elementului în vector și nu adresa de memorie, care se obține prin înmulțirea indexului obținut cu dimensiunea în octeți a unui element al matricii (sau tipul de bază al matricii).

Similar pentru o matrice tridimensională `B[M][N][Q]` se obține o formulă de forma:

```
addr(B[i][j][k]) = B + ((i*N +j)*Q + k)*sizeof(element) =
                     = B + (i*N*Q + j*Q + k)*sizeof(element)
```
(4)

iar pentru o matrice cvadruplu-dimensională `C[M][N][Q][P]` formula este:

```
addr(C[i][j][k][l]) = C + (((i*N +j)*Q + k)*P +l)*sizeof(element) =
                    = C + (i*N*Q*P + j*Q*P + k*P + l)*sizeof(element)
```
(5)

Se poate observa un model și există o formulă generică pentru calculul *offset*-ului în memorie pentru o matrice multidimensională $A{\times}B{\times}C...Y{\times}Z$ (sintaxă C: `M[A][B][C]...[Y][Z]`) pentru un elementul `M[a][b][c]...[y][z]` (în formulă sunt notați cu litere mici indecșii și cu majuscule valoarea maximă corespunzătoare acestora):

```
addr(M[a][b][c]...[y][z]) =
   = M + ((...((a*B + b)*C + c)...)*Y + y)*Z + z) *sizeof(element)
```

În formula pentru ordonarea *row-major*, indecșii sunt utilizați începând din stânga pe parcursul efectuării calculului pentru determinarea adresei unui element.

În mod convenabil matricile ordonate *row-major* fi văzute ca vectori de vectori. Fie definiția C pentru și un vector unidimensional `B` de tipul vector unidimensional `vect` de tip `float` (virgula mobilă simplă precizie) echivalentă cu definiția matrici bidimensionale de 4x4 elemente de tip `float`:

```
typedef float vect[4];
vect W[4];                                  // echivalent cu "float W[4][4];"
```

Variabila `W` este definită ca un tablou unidimensional (vector). Elementele sale individuale se întâmplă să fie vectori. Formula de calcula pentru adresa unui element dintr-o vector unidimensional este:

```
Addr(element) = Baza + Index*sizeof(element)
```

În acest caz `sizeof(element)` este 16, deoarece fiecare element al vectorului `W` este un vector de 4 variabile de tip `float` (`4*sizeof(float)=16`). Utilizând formula de mai sus se obține de fapt adresa de început a fiecărui rând din matricea 4x4 echivalentă.

Odată obținută adresa de bază a unui rând, se poate aplica din nou aceeași formulă pentru a obține adresa unui anumit element al vectorului unidimensional care constituie rândul. Deși acest lucru nu afectează rezultatul final, din punct de vedere conceptual, este probabil puțin mai ușor să se efectueze o secvență de calcule unidimensionale, mai degrabă decât de un singur calcul complex multidimensional. Diferența între cele două abordări se poate observa în cele două forme ale formulelor de calcul pentru matricea tridimensională `B` și cea cvadruplu-dimensională `C` de mai sus (ecuațiile (4) și (5)).

- ***Ordonarea matricilor după coloană (column-major)***

Ordonarea *column-major* este o altă funcție utilizată frecvent pentru a calcula adresa unui element al unei matrici. MATLAB stochează datele utilizând schema *column-major* (ordonarea în funcție de coloană), așa cum limbajul Fortran stochează matricile. MATLAB folosește această convenție deoarece inițial a fost scris în Fortran. MATLAB stochează intern elemente tablourilor de date începând cu prima coloană, apoi elemente celei de-a doua coloane și așa mai departe, până la ultima coloană [5].

În ordonarea *column-major*, indexul ce mai din dreapta este actualizat primul pe măsură ce se parcurg locații consecutive în memorie.

Formulele pentru calcularea adresei unui element matrice atunci când se utilizează ordonarea *column-major* este similară cu cea pentru ordonarea majoră a rândurilor: pur și simplu se inversează ordinea indecșilor și a dimensiunilor acestora în formula de calcul:

Pentru o matrice *column-major* cu două dimensiuni X[M][N], formula de clacul pentru adresa unui element este:

```
addr(X[i][j]) = X + (j*M + i)*sizeof(element)
```

Pentru o matrice tridimensională `Y[M][N][Q]` se obține formula:

```
addr(Y[i][j][k]) = Y + ((k*M + j)*N + i)*sizeof(element)
```

iar pentru o matrice cvadruplu-dimensională `C[M][N][Q][P]` formula este:

```
addr(Z[i][j][k][l]) = Z + (((l*M + k)*N + j)*Q + i)*sizeof(element)
```

- ***Alternative pentru organizarea matricilor***

Una din alternativele pentru stocarea pentru matrici dense este utilizarea de vectori Iliffe, care în mod obișnuit stochează elemente de pe același rând într-o zonă de date continuă (similar cu ordonarea *row-major*), dar nu și rândurile în sine. Sunt folosite de exemplu în: Java, C#/.Net și Swift.

Un vector Iliffe, este o structură de date utilizată pentru a implementa tablourilor multidimensionale. Un vector Iliffe pentru o tablou (matrice) `n`-dimensional (unde `n` este minim `2`) constă dintr-un vector (sau tablou unidimensional) de pointeri către o un tablou (`n-1`)-dimensional. Acestea sunt adesea folosite pentru a evita necesitatea operațiunilor de multiplicare costisitoare atunci când se efectuează calculul adresei pentru un element al unei matrici. Acestea pot fi, de asemenea, utilizate pentru a implementa matrici de forme neregulate. Structura de date este numită după John K. Iliffe.

Alte alternative sunt utilizarea listelor de liste, de exemplu, în Python sau utilizarea tabelelor de tabele, de exemplu, în Lua.

3.3.1.4. Declararea tablourilor multidimensionale

Un tablou bidimensional (matrice) de dimensiune *M*×*N*, va avea *M*N* elemente și va ocupa un spațiu de *M*N*sizeof(element)* octeți în memorie. Pentru a aloca spațiu de stocare pentru o matrice trebuie rezervată această cantitate de memorie și pentru aceasta există mai multe moduri diferite de a realiza această sarcină.

După cum a fost prezentata anterior operatorul `dup` poate fi utilizat pentru alocarea spațiului de stocare, acesta multiplicând de un număr de ori (ce precede directiva `dup`) *expresia* specificată în paranteze. Acest operator `dup` permite nu doar unul, ci mai multe elemente în paranteze și permite inclusiv apariții suplimentare ale operatorului `dup`.

De exemplu în următoarea afirmație:

```
A1 REAL4 4 dup (4 dup (?))
```

primul operator DUP multiplică de patru ori conținutul din paranteze. În interiorul parantezelor, expresia „4 dup (?)” va aloca spațiu pentru patru numere reale în simplă precizie, care va ocupa 4*4=16 octeți (SIZEOF REAL4=4). Patru copii ale celor patru numere reale ocupă 64 octeți, numărul necesar pentru o matrice 4×4. Desigur, pentru a rezerva spațiu de stocare pentru această matrice, se poate utiliza la fel de ușor o declarație de forma:

```
A2 REAL4 16 dup(?)
```

În oricare dintre cele două variante, asamblorul va aloca o zonă de 64 octeți adiacenți în spațiul de memorie. În ceea ce privește procesorul x86, nu există nicio diferență între aceste două forme. Pe de altă parte, declarația pentru variabila A1 sugerează programatorului o utilizare a respectivei zone de memorie ca o matrice 4×4, spre deosebire de declarația pentru variabila A2 care sugerează un vector de 16 elemente.

Conceptul poate fi ușor extins la matrici de ordin superior, de exemplu declarația C pentru o matrice cu trei dimensiuni: „float A3[5][6][8];” de numere reale în simplă precizie se traduce în asamblare ca:

```
A3 REAL4 5 dup (6 dup (8 dup (?)))
```

La fel ca în cazul matricilor unidimensionale (vectorilor) se poate inițializa fiecare element al matricei la o anumită valoare înlocuind caracterul '?' cu o anumită valoare. De exemplu, pentru a inițializa matricea de mai sus astfel încât fiecare element să conțină valoarea 3.14, se poate utiliza declarația:

```
A3 REAL4 5 dup (6 dup (8 dup (3.14)))
```

Dacă fiecare element al matricii trebuie inițializat la o valoare diferită, va trebui să introdusă fiecare valoare individual. Dacă dimensiunea unui rând este suficient de mică, cel mai bun mod de a aborda această sarcină este plasarea datelor pentru fiecare rând al matricii pe propria linie, ca de exemplu în declarația pentru matricea 4×4:

```
A4 REAL4 0.0, 0.1, 0.2, 0.3
   REAL4 1.0, 1.1, 1.2, 1.3
   REAL4 2.0, 2.1, 2.2, 2.3
   REAL4 3.0, 3.1, 3.2, 3.3
```

Din punct de vedere al asamblorului nu contează cum sunt împărțite liniile, dar declarația de mai sus este mult mai ușor de identificat ca o matrice 4×4 decât următoarea declarație care generează exact aceleași date:

```
A4 REAL4 0.0,0.1,0.2,0.3,1.0,1.1,1.2,1.3,2.0,2.1,2.2,2.3,3.0,3.1,3.2,3.3
```

Desigur, o matrice matrice mare (cu rânduri foarte mari sau o matrice cu mai multe dimensiuni) există puține speranțe pentru a genera o declarație sugestivă. Mai ales în ceste cazuri sunt utile comentariile care explică în detaliu intenția programatorului.

3.3.1.5. Accesarea elementelor tablourilor multidimensionale

Pentru a accesa un element al matrici trebuie obținută adresa elementului matricii. Aceasta se poate realiza prin implementarea calcului conform formulelor prezentate anterior pentru ordonările *row-major* respectiv *column-major*. Instrucțiunile MOV, ADD și (I)MUL în combinație cu modurile de adresare ale procesorului pot fi utilizate pentru calculul deplasamentelor elementelor unei matrici multidimensionale. Unele compilatoare (de exemplu gcc) nu utilizează instrucțiunile (i)mul ci realizează, pentru matrici de dimensiuni mici, operația de înmulțire printr-o secvență de deplasări utilizând instrucțiunea SHL/SAL și adunări.

Fie matrici `M` de dimensiune `10×8` de numere întregi fără semn pe 32 biți și doi indecși `i` și `j` (pe 64 biți), definiți în zone de date prin următoarele declarații:

```
M DWORD 10 dup (15 dup (?) )
i QWORD ?
j QWORD ?
```

Pentru a realiza în program o operație de inițializare a unui element cu o valoare oarecare, (operație similara atribuirii C/C++: `M[i][j]=100;`), se poate utiliza secvența de instrucțiuni:

```
imul rax, QWORD PTR i, 15              ; 15 = nr de coloane
add rax, j
lea rdx, M
mov DWORD PTR [rdx+rax*TYPE M], 100    ; TYPE M = SIZEOF DWORD = 4
```

În secvența de cod de mai sus se calculează în registrul `RAX` deplasamentul elementului în cadrul matricii și se folosește adresarea bazată și indexată pentru a accesa locația la case se găsește acesta. Utilizarea modului de adresare cu două registre (adresarea bazată și indexată) nu este obligatorie. Deși adresarea utilizând două registre ar părea un mod natural pentru accesarea matricilor bidimensionale, acesta nu este scopul acestui mod de adresare.

În cazul în care numărul de coloane este putere al lui doi sau valori care se pot descompune în puține valori ale lui doi operațiunea de înmulțire se poate înlocui cu secvențe scurte de deplasări și adunări. Astfel secvența de mai sus se poate implementa și sub forma:

```
mov rax, i                  ; RAX = i
mov rdx, rax                ; RDX = i
shl rax, 4                  ; RAX = RAX<<4 = RAX*16 = 16*i
sub rax, rdx                ; RAX = RAX-RDX = 16*i-i = 15*i
add rax, j
lea rdx, M
mov DWORD PTR [rdx+rax*TYPE M], 100    ; TYPE M = SIZEOF DWORD = 4
```

3.3.2. Structuri de date

Al doilea tip de date compozit este *structura*. În timp ce un tablou este omogen, ale cărui elemente sunt toate de același tip, elementele dintr-o structură pot fi de orice tip. Tablourile permit accesul la un anumit element prin intermediul unui index care este număr întreg. Structurile permit accesul la un element după numele acestuia denumit câmp (engl. *field*). Scopul unei structuri este acela de a permite încapsularea de date diferite, dar legate în mod logic între ele, grupate într-un singur pachet.

În asamblare o astfel de tip de structură se poate crea folosind declarația `STRUCT` (pentru compatibilitate cu versiunile anterioare `UASM` acceptă și declarația `STRUC`). Sintaxa unei declarații `STRUCT` este:

```
nume_tip STRUCT [aliniament]
       ; declaratii campuri/fields
nume_tip ENDS
```

O declarație pentru o structură se termină cu declarația `ENDS` (de la engl. *end structure*). Eticheta dinainte de declarației `ENDS` trebuie să fie aceeași cu cea care precede declarația `struct`. Numele câmpurilor din structură trebuie să fie unice, cu alte cuvinte un nume nu poate să apară de două sau mai multe ori în cadrul aceleași structuri. Cu toate acestea, toate numele câmpurilor

sunt locale pentru structura respectivă și pot fi reutilizate ca variabile globale sau câmpuri în cadrul altor structuri. Dacă parametru numeric opțional `aliniament` este prezent, atunci deplasamentele câmpurilor structurii vor fi multiplu de aceasta valoare numerică imediată.

Directiva `STRUCT` definește doar un tip de structură. Nu rezervă spațiu pentru o variabilă de tipul structurii respective. Pentru a rezerva efectiv spațiu de stocare, trebuie declarată o variabilă utilizând numele structurii ca tipul variabilei respective, de forma:

```
nume_variabila nume_tip {⟦valori_initiale⟧}
```

unde între acolade opțional se pot afla valori inițiale pentru câmpurile structurii.

Fie, de exemplu, structura:

```
RECTANGLE STRUCT
        x1 WORD ?
        y1 WORD ?
        x2 WORD ?
        y2 WORD ?
        attr BYTE ?
RECTANGLE ENDS
```

Declarația pentru o variabilă de tipul `RECTANGLE` va avea forma:

```
chenar RECTAGLE {}
```

Asamblorul va aloca variabilei o zonă continuă de memorie în zona de date astfel încât dacă eticheta `chenar` reprezintă adresa de bază (de început) a structurii, primul element acesteia `x1` se va găsesc la locația `[chenar+0]`, al doilea element `y1` la adresa `[chenar+2]` (`SIZEOF WORD=2`) și așa mai departe până la câmpul `attr` care se va găsi la locația `[chenar+8]`.

Pentru a accesa un element al unei structuri, trebuie determinat deplasamentul față de adresa de începutul a structurii. De exemplu, câmpul `attr` din variabila `chenar` se află la un *offset* de `8` față de adresa de bază variabilei `chenar`. Prin urmare, este posibil stocarea valorii din acest câmp în registru `AL`, de exemplu, folosind instrucțiunea:

```
mov al, chenar[8]
```

Asamblorul permite un mod de acces la câmpurile unei structuri mult mai facil folosind același mecanism ca și limbajul C: operatorul „`.`” (punct). Pentru a realiza aceeași operațiune de stocare a valorii câmpului attr în registrul AL se poate utiliza sintaxa:

```
mov al, chenar.attr
```

Acestă sintaxă este mult mai lizibilă și cu siguranță mai ușor de utilizat. Utilizarea operatorului „`.`” nu introduce un nou mod de adresare, asamblorul adaugă pur și simplu la adresa de bază a variabilei `chenar` deplasamentul până la câmpul `attr` (`8`) pentru a obține *offset*-ul necesar pentru a fi codificat în instrucțiune.

Se pot, de asemenea, specifica valori inițiale implicite atunci când se creează o structură. Pur și simplu în locul valorilor nedeterminate (specificate prin „`?`”) se definesc valori predefinite pentru fiecare declarație a fiecărui câmp. Există două moduri diferite de a specifica o valoare inițială pentru câmpurile unei structuri, ca în declarația:

```
RECTANGLE STRUCT
        x1 WORD 1
        y1 WORD 1
        x2 WORD 40
        y2 WORD 10
```

```
        attr BYTE 0Eh
RECTANGLE ENDS
```

De fiecare dată când se definește o variabilă de tipul `RECTANGLE` printr-o definiție de forma:

```
chenar RECTAGLE {}
```

asamblorul va inițializa automat variabilele `chenar.x1` și `chenar.y1` cu valoarea `1`, `chenar.x2` cu valoarea `40`, `chenar.y2` cu valoarea `10` și `chenar.attr` cu valoarea hexazecimală `0Eh`. Acest lucru funcționează excelent în cazurile în care variabilele au de obicei aceleași valori inițiale (de exemplu *pointer*-i care trebuie inițializați cu `NULL`).

Inițializarea câmpurilor unei structuri se poate face și la declararea variabilei de tipul structurii respective. Acest lucru este ușor de realizat prin specificarea valorilor inițiale în interiorul acoladelor:

```
chenar1 RECTAGLE {0,1,20,20,0}
chenar2 RECTAGLE {10,11,33,45,15}
chenar3 RECTAGLE { , ,30,10,?}
```

Asamblorul inițializează valorile câmpurilor în ordinea în care acestea apar între paranteze. Tipul valorii de inițializare trebuie să se potrivească cu tipul câmpului corespunzător din definiția structurii – nu se inițializa un câmp de de tip `word` cu o valoare mai mare decât `65535`, de exemplu.

Nu este necesară inițializarea tuturor câmpurilor dintr-o structură. Dacă un câmp este necompletat, asamblorul `UASM` va utiliza valoarea implicită specificată în definiția structurii.

3.3.2.1. Structuri de structuri

Structurile pot conține alte structuri sau tablouri ca și câmpuri. Fie următoarele definiții:

```
COORD STRUCT
      x     WORD ?
      y     WORD ?
COORD ENDS
TRIANGLE STRUCT
      pt1   COORD <>
      pt2   COORD <>
      pt3   COORD <>
      color DWORD ?
TRIANGLE ENDS
```

Definiția de mai sus definește un o structură de tip `COORD` și o structură `TRIANGLE` care conține trei elemente de tip `COORD` cu numele `pt1`, `pt2`, `pt3` și o valoare `DWORD` pentru câmpul `culor`. La inițializarea unui obiect de tip `TRIANGLE`, primul inițializator corespunde câmpului de tipul `COORD` și nu câmpului cu coordonate `x`. Inițializări corecte pentru variabile de tip `TRIANGLE` sunt:

```
triunghi1 TRIANGLE {}
triunghi2 TRIANGLE { ,<>,{},?}
triunghi3 TRIANGLE {{10,2},{ ,7},{5,?},0}
```

Accesarea câmpurilor este similară limbajelor de nivel înalt, se utilizează operatorul „`.`” (punct) odată pentru a face referință la câmpul `pt1` și o a doua oară pentru a accesa câmpurile `x` sau `y` ale structurii `COORD`, ca în exemplele:

```
mov ax, triunghi1.pt1.x
mov triunghi2.pt3.y, 0
cmp triunghi2.pt1.x, 100
```

3.3.2.2. Pointeri la structuri

De multe ori este necesară referirea la o structură direct sau indirect folosind un pointer. Când se utilizează un pointer pentru a accesa câmpurile unei structuri, trebuie încărcat unul dintre regiștrii de uz general pe 64 de biți ai procesorului cu adresa structurii dorite. Fie următoarele declarații de variabile (și structura `COORD` aferentă):

```
COORD STRUCT
      x     WORD ?
      y     WORD ?
COORD ENDS
Point COORD {}
PointPTR QWORD Point
```

`PointPTR` conține adresa obiectului `Point` (adică este un *pointer* către acesta). Pentru a accesa câmpul `x` al obiectului `Point`, după cum s-a prezentat anterior, se poate utiliza sintaxa:

```
mov ax, Point.x
```

Deoarece o instrucțiune poate accesa o singură locație de memorie accesați un câmp prin intermediul unui *pointer*, trebuie neapărat făcută prin intermediul regiștrilor (modurile de adresare indirecte). Adresa structurii trebuie deci încărcată într-un registru de uz general pe 64 biți fie prin instrucțiunea `LEA`, fie prin copierea pointer-ului. Fie următoarea secvență de cod:

```
lea rbx, Point
mov ax, [rbx].x     ; Error: Symbol not defined : x
```

Cea de-a doua instrucțiune din secvența de mai sus va genera eroare. Întrucât numele câmpurilor sunt locale pentru o structură și este posibilă reutilizarea unui nume de câmp în două sau mai multe structuri, numele câmpurilor nu sunt unice și deci asamblorul nu poate determina deplasamentul doar pe baza numelui unui câmp. Când sunt accesați direct membrii unei structurii (de exemplu, „`mov ax, Point.x`”) nu există nici o ambiguitate, deoarece tipul variabilei `Point` poate fi verificat de către asamblor, `[rbx]` pe de altă parte, conține doar adresa și nu și tipul variabilei. De aceea, asamblorul nu poate, să decidă ce offset să utilizeze pentru câmpul `x`.

Această ambiguitate se poate rezolva prin specificarea în mod explicit a tipului variabilei. Probabil cel mai simplu mod de a face acest lucru este să specificați numele structurii ca pseudo-câmp (engl. *pseudo-field*). Un pseudo-câmp nu este un câmp al unei structuri ci este numele structurii în sine, care este specificat ca primul parametru al operatorului „`.`” (practic după primul „`.`” se specifică tipul pointer-ului) ca în secvențele:

```
lea rbx, Point
mov ax, [rbx].COORD.x
```

```
mov rsi, PointPTR
mov ax, [rsi].COORD.y
```

Prin specificarea numelui structurii, asamblorul va putea calcula corect deplasamentul (*offset*-ul) pentru câmpurile structurii respective.

Această sintaxa nu funcționează doar cu adresarea bazată ci cu oricare mod de adresare indirectă: bazată și/sau indexată și/sau cu deplasament. Fie un vectorul de șase structuri de tip

COORD (definită anterior) definit prin declarația:

```
hexagon COORD 6 dup ({})
```

Accesul la al patrulea (index 3) punct din vector, de exemplu, se poate realiza folosind adresarea bazată și cu deplasament:

```
lea rbx, hexagon
mov ax, [rbx + 3*SIZEOF COORD].COORD.x
```

Capitolul 4. Setul de instrucțiuni de bază x86-64

Setul de instrucțiuni x86-64 (cunoscut și sub denumirea de x64, x86_64, AMD64 sau Intel 64) este versiunea extinsă pe 64 biți a setului de instrucțiuni a procesoarelor x86 (cunoscut ulterior și sub denumirea x86-32). x86-64 extinde regiștrii de uz general la 64 biți, extinde și numărul acestora de la 8 (unele dintre ele având inițial funcționalități limitate sau fixe, de exemplu pentru gestionarea stivei) la 16 și oferă numeroase alte îmbunătățiri. Operațiile în virgulă mobilă sunt realizate prin instrucțiuni asemănătoare SSE2, iar regiștrii în stil x87/MMX nu sunt în general utilizați (dar sunt disponibili chiar și în modul pe 64 biți); în schimb, este utilizat un set de 32 de regiștri de tip vector, 128 biți (fiecare putând stoca unul sau două numere în dublă precizie sau unul sau patru numere în simplă precizie sau diferite formate de numere întregi). În modul pe 64 biți, instrucțiunile sunt modificate pentru a suporta operanzi pe 64 biți și modul de adresare pe 64 biți. În modul de compatibilitate pot fi rulate aplicații utilizator pe 16 și 32 biți, fără a fi necesară modificarea acestora, coexistând cu aplicații pe 64 biți dacă sistemul de operare pe 64 biți le acceptă [4].

Instrucțiunile de uz general implementează instrucțiuni de bază precum: transferul datelor, funcții aritmetice și logice, controlul fluxului de execuție și operațiuni cu șiruri, pe care programatorii le folosesc în mod obișnuit pentru a scrie aplicații software pentru a fi rulate pe procesoare Intel 64 și IA-32. Acestea operează cu date conținute în memorie, în regiștrii de uz general (`RAX`, `RBX`, `RCX`, `RDX`, `RDI`, `RSI`, `RBP`, `RSP` și `R8–15`) și în registrul `EFLAGS`. De asemenea, operează cu adrese conținute în memorie sau în regiștrii de uz general .

Acest grup de instrucțiuni include transferul de date, aritmetica întreagă binară, aritmetica zecimală, operațiile logice, operațiile de deplasare și rotire, instrucțiuni pe `bit` și `byte`, controlul programului, operații cu șiruri, controlul programului și sub-grupul de instrucțiuni diverse.

4.1. Sintaxa instrucțiunilor

Fiecare instrucțiune are o sintaxă bazată pe mnemonici folosite de asamblori pentru a specifica operația și operanzii care vor fi utilizați ca data sursă și/sau data destinație (rezultat). Figura 4.1 prezintă un exemplu de sintaxă a unei mnemonici pentru o instrucțiune de comparație (`CMP`) [6]. În acest exemplu, mnemonica `CMP` este urmată de doi operanzi, un registru pe 64 biți sau un operand de memorie și un operand imediat pe 8 biți.

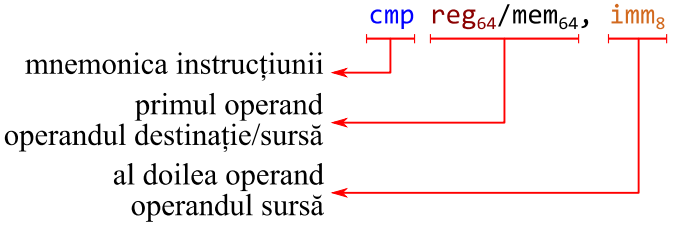

Figura 4.1: Exemplu de sintaxă a unei instrucțiuni în limbaj de asamblare.

În majoritatea instrucțiunilor cu doi operanzi, primul operand (cel din stânga) este atât un operand sursă, cât și operand destinație. Al doilea operand (cel din drepta) servește doar ca sursă. Cu excepția cazului în care este altfel specificat, cuvântul registru înseamnă un registru de uz general (GPR[1]). Mai multe instrucțiuni afectează biții de stare din registrul `RFLAGS`.

4.2. Instrucțiuni de transfer al datelor

1 GPR – din engl. General Purpose Register

4.2.1. Instrucțiunea MOV.

Forma generală a instrucțiunii MOV (Move) este:

```
mov    dest, sursa
```

Instrucțiunea MOV transferă (copiază) valoarea celui de-al doilea operand (operand sursă) în primul operand (operandul destinație). Operandul sursă poate fi o valoare imediată, registru de uz general sau locație de memorie; registrul de destinație poate fi un registru de uz general sau o locație de memorie. Ambii operanzi trebuie să aibă aceeași dimensiune, care poate fi un octet (8 biți), un cuvânt (16 biți), un dublu-cuvânt (32 biți) sau un cvadruplu-cuvânt (64 biți).

Instrucțiunea MOV are câteva forme uzuale:

```
mov    regx, regx
mov    memx, regx
mov    regx, memx
mov    memx, immy                                  ; Y = 8/16/32
mov    regx, immx
mov    memx, AL/AX/EAX/RAX
mov    AL/AX/EAX/RAX, memx                         ; X = 8/16/32/64
```

Se poate observa studiind formele de mai sus ca singura formă care acceptă o valoare imediată pe 64 biți este cea în care operandul destinație este un registru pe 64 biți. În cazul în care operandul destinație este locație de memorie valoarea imediată codată în instrucțiune poate avea cel mult 32 biți. Astfel, dacă locația de memorie este pe 64 biți valoarea imediată pe 32 biți va fi extinsă cu semn la 64 biți.

Există un detaliu foarte important de remarcat despre instrucțiunea MOV. Nu există posibilitatea de a copia o locație de memorie într-o altă locație de memorie (ambii operanzi nu pot fi locații de memorie). Nu există nici o formă a instrucțiunii MOV care să permită codificarea a două adrese de memorie în aceeași instrucțiune.

Operanzii instrucțiunii MOV pot fi BYTE, WORD, DWORD sau QWORD. Ambii operanzi trebuie să aibă aceeași dimensiune sau asamblorul va genera o eroare în timpul asamblării programului. Acest lucru se aplică operanzilor de memorie și operanzilor regiștri. Dacă, de exemplu, se încearcă copierea unei variabile, var_B, declarată folosind BYTE în registrul AX, asamblorul va genera o eroare.

Asamblorul extinde automat datele imediate (constante) la dimensiunea operandului destinație (cu excepția cazului în care acestea sunt prea mari pentru a se încadra în operandul de destinație, ceea ce va genera eroare). Este de reținut faptul că valorile imediate pot fi transferate într-o locație de memorie. Se aplică aceleași reguli privind dimensiunea. Cu toate acestea, asamblorul nu poate determina dimensiunea anumitor operatori de memorie. De exemplu, instrucțiunea:

```
mov    [rbx], 7
```

stochează o valoare de 8, 16, 32 sau 64 biți? asamblorul nu poate determina dimensiunea nici unuia dintre operanzi, și va genera mesajul de eroare „invalid instruction operands” (operanzi nepermiși pentru instrucțiune). Această problemă nu există apare când se transferă o valoare imediată într-o variabilă declarată în program. De exemplu, dacă var_b este declarată ca o variabilă pe 8 biți (BYTE), asamblorul poate stoca un număr pe de 8 biți în var_b pentru instrucțiunea:

```
mov     var_b, 7
```

Doar acei operatori de memorie care implică indicatori fără operanzi variabili suferă de

această problemă. Soluția este să se specifice explicit asamblorului dacă operandul este un octet (BYTE), un cuvânt (WORD), un dublu-cuvânt (DWORD) sau cvadruplu-cuvânt (QWORD). Acest lucru poate fi realizat utilizând operatorul PTR, obținând formele:

```
mov    byte ptr [rbx], 7
mov    word ptr [rbx], 7
mov    dword ptr [rbx], 7
mov    qword ptr [rbx], 7
```

Instrucțiunile MOV nu afectează nici un *flag*. În special, procesorul păstrează valorile *flag*-urilor în timpul execuției unei instrucțiuni de tip MOV.

4.2.2. Instrucțiunile CMOVcc

Forma generală a instrucțiunilor CMOV*cc* (Conditional Move) este:

```
cmovcc  dest, sursa
```

Instrucțiunile CMOV*cc* sunt un grup de instrucțiuni care verifică starea indicatorilor de condiție din registrul RFLAGS și efectuează o operație de transfer dacă *flag*-urile sunt într-o stare specificată. Aceste instrucțiuni pot fi utilizate pentru a muta o valoare de 16, 32 sau 64 biți din memorie într-un registru de uz general sau între regiștrii de uz general. Starea *flag*-ului testat este specificată cu un cod de condiție (*cc*) asociat cu instrucțiunea. Dacă condiția nu este îndeplinită, transferul (copierea) nu este efectuată și execuția continuă cu instrucțiunea imediat următoare instrucțiunii CMOV*cc*.

Forme specifice ale instrucțiunii CMOV*cc* sunt (unde X=16,32,64):

```
cmovcc  regx, regx
cmovcc  regx, memx                                  ; X = 16/32/64
```

Spre deosebite de instrucțiunea MOV, operanzii instrucțiunii CMOV*cc* pot fi doar WORD, DWORD sau QWORD, dar *nu pot fi* BYTE. Ambii operanzi trebuie să aibă aceeași dimensiune altfel asamblorul va genera o eroare în timpul asamblării programului.

Tabelul 4.1 prezintă mnemonicele pentru instrucțiunile CMOV*cc* și condițiile testate pentru fiecare instrucțiune [7]. Acronimul în engleză al codului de condiție (cc) este anexat la literele „CMOV” pentru a forma mnemonica pentru instrucțiunile CMOV*cc*. Instrucțiunile enumerate în tabel ca perechi (de exemplu, CMOVA/CMOVNBE) sunt nume alternative pentru aceeași instrucțiune. Asamblorul oferă aceste nume alternative pentru a facilita citirea mai ușoară a programelor.

Instrucțiunile CMOV*cc* sunt utile pentru optimizarea construcțiilor IF mici. Software-ul poate verifica dacă instrucțiunile CMOV*cc* sunt acceptate verificând informațiile caracteristice ale procesorului cu instrucțiunea CPUID.

Tabelul 4.1. Instrucțiunile de transfer condiționat `CMOVcc`.

	Mnemonica	*Starea FLAG-urilor*	*Descriere*
Evaluări fără semn	`CMOVA/CMOVNBE`	`(CF` *or* `ZF) = 0`	Above / Not Below or Equal
	`CMOVAE/CMOVNB`	`CF = 0`	Above or Equal / Not Below
	`CMOVNC`		Not Carry
	`CMOVB/CMOVNAE`	`CF = 1`	Below / Not Above or Equal
	`CMOVC`		Carry
	`CMOVBE/CMOVNA`	`(CF` *or* `ZF) = 1`	Below or Equal / Not Above
	`CMOVE/CMOVZ`	`ZF = 1`	Equal / Zero
	`CMOVNE/CMOVNZ`	`ZF = 0`	Not Equal / Not Zero
	`CMOVP/CMOVPE`	`PF = 1`	Parity / Parity Even
	`CMOVNP/CMOVPO`	`PF = 0`	Not Parity / Parity Odd
Evaluări cu semn	`CMOVG/CMOVNLE`	`((SF` *xor* `OF)` *or* `ZF) = 0`	Greater / Not Less or Equal
	`CMOVGE/CMOVNL`	`(SF` *xor* `OF) = 0`	Greater or Equal / Not Less
	`CMOVL/CMOVNGE`	`(SF` *xor* `OF) = 1`	Less / Not Greater or Equal
	`CMOVLE/CMOVNG`	`((SF` *xor* `OF)` *or* `ZF) = 1`	Less or Equal / Not Greater
	`CMOVO`	`OF = 1`	Overflow
	`CMOVNO`	`OF = 0`	Not Overflow
	`CMOVS`	`SF = 1`	Sign (negative)
	`CMOVNS`	`SF = 0`	Not Sign (non-negative)

4.2.3. Instrucțiunea XCHG.

Instrucțiunea `XCHG` (Exchange Register/Memory with Register) interschimbă două valori. Forma sa generală este:

```
xchg   operand1, operand2
```

Forme specifice ale instrucțiunii `xchg` sunt:

```
xchg   regX, memX
xchg   memX, regX
xchg   regX, regX                          ; X = 8/16/32/64
xchg   AX/EAX/RAX, memY
xchg   memY, AX/EAX/RAX                    ; Y = 16/32/64
```

Cea de-a treia formă conține de fapt trei forme ale instrucțiunii, și sunt forme speciale ale celei de-a doua forme care interschimbă valoarea din registrul `AX`, `EAX` sau `RAX` cu un alt registru de 16, 32 sau 64 biți.

Se poate recunoaște deja un model: familia de procesoare x86 oferă adesea versiuni mai scurte și mai rapide ale instrucțiunilor care utilizează registrul acumulator. Prin urmare, ar trebui pe cât posibil să se aranjeze calculele astfel încât să se utilizeze registrul acumulator cât mai

mult. Instrucțiunea XCHG este un exemplu perfect, codificarea sa pentru forma care interschimbă regiștrii de 16 biți este mai scurtă cu un octet.

Nu contează ordinea operanzilor instrucțiunii XCHG. Astfel, instrucțiunea:

```
xchg   mem, reg
```

va produce același rezultat ca:

```
xchg   reg, mem
```

Majoritatea asambloarelor moderne vor genera automat codul mai scurt. De exemplu instrucțiunea:

```
xchg   ax, reg16
```

codificată pe 2 octeți va înlocui varianta mai lungă, codificată pe 3 octeți:

```
xchg   reg16, ax
```

Operanzii trebuie să fie de aceeași dimensiune. Instrucțiunea XCHG nu modifică indicatorii de condiție (*flag*-urile).

Când unul din operanzii instrucțiunii XCHG este locație de memorie, semnalul LOCK al procesorului este automat setat. Această instrucțiune este astfel utilă pentru implementarea semafoarelor sau a structurilor de date similare pentru sincronizarea proceselor.

4.2.4. Instrucțiunea LEA

Instrucțiunea LEA (Load Effective Address) este o instrucțiune folosită pentru încărcarea adreselor (lucru cu pointeri). Instrucțiunea LEA ia forma:

```
lea    dest, sursa
```

Forme specifice ale instrucțiunii lea sunt:

```
lea    reg16, mem16
lea    reg32, mem32
lea    reg64, mem64
```

Instrucțiunea LEA încarcă un registrul de uz general 16, 32 sau 64 biți specificat cu *adresa efectivă* a locației de memorie specificate. Adresa efectivă este adresa finală de memorie obținută după toate calculele modului de adresare. Fie următoarele instrucțiuni:

```
lea    rax, [rbx]
lea    rbx, [rbx+5]
lea    rax, 5[rbx]
lea    rbx, [rbp+rsi+7]
lea    rax, [rdi-200h]
```

Prima instrucțiune „`lea rax, [rbx]`” copiază adresa expresiei [RBX] în registrul acumulator RAX. Întrucât adresa efectivă este valoarea din registrul RBX, această instrucțiune copiază valoarea din RBX în RAX. Această instrucțiune nu este foarte utilă, deoarece o instrucțiune MOV poate face același lucru, chiar mai rapid.

Instrucțiunea „`lea rbx, [rbx+5]`” copiază adresa efectivă a variabilei [RBX+5] în registrul RBX. Deoarece această adresă efectivă este egală cu suma valoarii curente a registrului RBX și a valorii 5, această instrucțiune LEA adună efectiv numărul 5 la registrul RBX. Instrucțiunea ADD realizează adunarea numărului 5 la registrul RBX, astfel încât din nou, instrucțiunea LEA este de prisos în acest scop.

Începând cu cea de-a treia instrucțiune („`lea rax, 5[rbx]`”), instrucțiunea LEA începe să

să își arate cu adevărat valoarea. Aceasta copiază adresa locației de memorie „5[RBX]” în registrul acumulator RAX; adică, adaugă 5 cu valoarea din registrul RBX și mută suma în RAX. Acesta este un exemplu excelent al modului în care instrucțiunea LEA poate fi utilizată pentru a efectua o operație MOV și una ADD cu o singură instrucțiune LEA. În UASM sintaxa „5[RBX]” sau „[RBX+5]” sunt echivalente.

Ultimele două instrucțiuni din exemplele de mai sus, oferă exemple suplimentare de utilizări pentru instrucțiunea LEA care sunt mai eficiente decât omologii lor MOV / ADD.

Pe procesoarele x86-64, putem utiliza modurile de adresare indexate și scalate pentru a se realiza multiplicări cu 2, 4 sau 8, precum și adunarea de regiștri și deplasament simultan. Intel sugerează cu fermitate utilizarea instrucțiunii LEA, deoarece este mult mai rapidă decât o secvență de instrucțiuni care calculează același rezultat, cu diferența că operația aritmetică efectuată (adunare și sau înmulțire) nu va seta *flag*-urile [7].

4.2.5. Instrucțiunile PUSH și POP

Instrucțiunile PUSH și POP manipulează datele din stiva hardware x86-64. Acestea transferă datele către și din stivă. Forme ale acestor instrucțiuni sunt:

```
push   regX/memX
pop    regX/memX                                   ; X = 16/64
push   immY                                        ; Y = 8/16/32
```

Instrucțiunea PUSH micșorează indicatorul stivei (conținut în registrul RSP) cu 2 sau cu 8 funcție de dimensiunea operandului, apoi copiază operandul sursă în vârful stivei (a se vedea Figura 4.2). Instrucțiunea operează pe locații de memorie, valori imediate și regiștri. Instrucțiunea PUSH este folosită în mod obișnuit pentru a plasa parametrii pe stivă înainte de a apela o procedură. Poate fi folosit și pentru a rezerva spațiu pe stivă pentru variabile temporare.

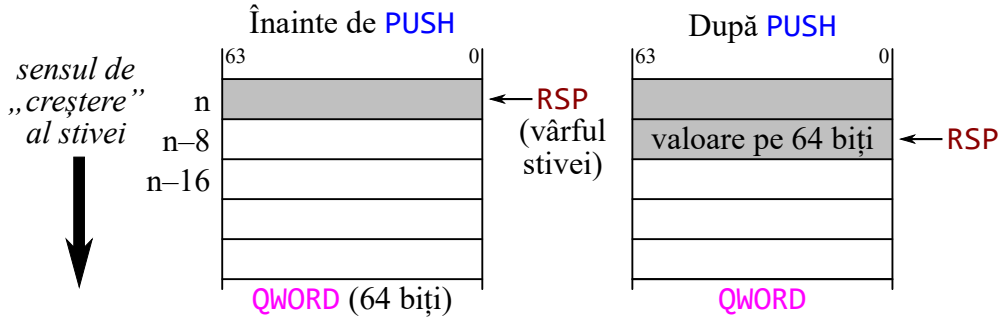

Figura 4.2. Funcționarea instrucțiunii PUSH.

Dacă operandul sursă al instrucțiunii PUSH este o valoare imediată, o valoare pe 64 biți (cu extinderea semnului) este depusă pe stivă.

Este o practică obișnuită utilizarea instrucțiuni PUSH pentru a transmite parametrii (prin stivă) către funcții și subrutine. Secvența de instrucțiuni tipică folosită la începutul unei subrutine arată astfel:

```
push   rbp          ; salvare valoarea curenta a reg. RBP pe stiva
mov    rbp, rsp     ; seteaza EBP = cadrului stivei (stack-frame)
sub    rsp, N       ; alocare spatiu pentru variabilele locale
```

Registrul RBP este utilizat ca indicator pentru cadrul stivei (din engl. *Stack-frame*) – o adresă de bază a zonei de stivă utilizată pentru parametrii transferați către subrutine și variabile locale. Compensările pozitive ale cadrului stivei indicat de RBP oferă acces la parametrii subrutinei în timp ce compensările negative accesează variabilele locale. Această tehnică permite

crearea subrutinelor imbricate.

De obicei, atunci când este apelată o procedură, stiva conține următoarele patru componente:

- parametrii transmiși către procedura apelată (încărcați de codul apelant);
- adresa de revenire (creată de instrucțiunea CALL);
- tablou de pointeri la *stack-frame*-uri (pentru stivuirea *stack-frame*-urilor procedurilor sub-apelate), care sunt utilizate pentru a accesa variabilele locale ale unor astfel de proceduri;
- variabilele locale utilizate de procedura apelată.

Toate aceste date se numesc *stack-frame*. Pentru a gestiona aceste componente ale unui *stack-frame,* în primul rând, valoarea curentă a registrului RBP este salvată pe stivă. Valoarea curentă a registrului RSP (în acest moment) este un indicator al cadrului pentru procedura curentă: compensările pozitive din acest indicator (RBP) oferă acces la parametrii trimiși procedurii, iar compensările negative oferă acces la variabilele locale care vor fi utilizate ulterior. Pe parcursul execuției procedurii, valoarea pointerului de cadru care este stocată în registrul RBP, va conține practic o copie a valorii vârfului stivei la stabilirea acestui cadru, care nu va suferi modificări, indiferent de modificările ulterioare ale valorii registrului RSP sau de apelul instrucțiunilor PUSH, POP, CALL sau RET.

La revenirea din procedură se eliberează memoria alocată pentru variabilele locale și pentru pointerul la *stack-frame*-uri. Aceasta se realizează prin următoarele două etape: în primul rând, valoarea inițială a vârfului stivei este copiată din registrul RBP în registrul RSP; în al doilea rând, registrul RBP este descărcat de pe stivă, restabilind valoarea anterioară a *stack-frame*-ului (valoarea registrului RBP).

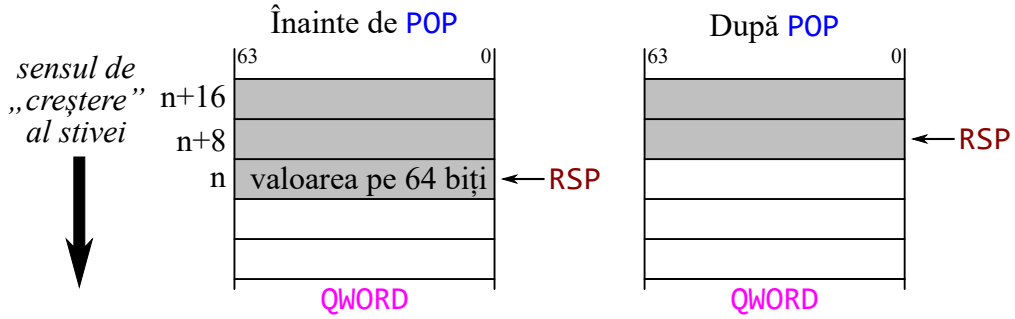

Figura 4.3. Funcționarea instrucțiunii POP.

Instrucțiunea POP copiază un cuvânt sau cvadruplu-cuvânt (patru cuvinte) din locația de memorie indicată de registrul RSP (vârful stivei) într-un registru sau locație de memorie specificată. Apoi, registrul RSP este incrementat cu 2, respectiv 8. După operația POP, RSP indică noul vârf al stivei (a se vedea Figura 4.3).

Instrucțiunea „push rsp” depune în vârful stivei valoarea registrului RSP așa cum exista înainte de executarea instrucțiunii. Dacă o instrucțiune PUSH folosește un operand de memorie în care este utilizat registrul RSP, adresa operandului este calculată înainte de decrementarea registrului RSP.

Funcționarea instrucțiunilor PUSH și POP poate fi descrisă de algoritmii:

```
push (16 biti):
       RSP = RSP - 2
       [RSP] = 16-bit operand ; stocheaza valoarea la adresa RSP
pop  (16 biti):
       16-bit operand = [RSP]
```

```
            RSP = RSP + 2
push  (64 biti):
            RSP = RSP - 8
            [RSP] = 64-bit operand
pop   (64 biti):
            64-bit operand = [RSP]
            RSP = RSP + 8
```

Se pot observa trei lucruri despre stiva hardware x86-64. În primul rând, este întotdeauna adresată prin intermediul registrului RSP. În al doilea rând, stiva crește „în jos" în memorie – depunerea (adăugarea) unei valori în stivă va scădea valoarea lui RSP și invers extragerea unei valori va crește valoarea registrului RSP. Adică, pe măsură ce depunem valori pe stivă, procesorul le stochează în locații de memorie din ce în ce mai mici (inferioare). În cele din urmă, indicatorul stivei hardware RSP conține întotdeauna adresa valorii din vârful stivei (ultima valoare depusă pe stivă).

Motivul pentru care stiva crește către adrese inferioare datează din aprilie 1974, când Intel a lansat procesorul pe 8 biți 8080, care a influențat designul primului procesor din seria x86, 8086. Arhitectul procesorului 8080, Stanley Mazor, explică de ce s-a ales ca stiva să crească în acest fel: „Indicatorul stivei a fost ales să crească «în jos» (stiva înaintând spre zonele de memorie inferioare) pentru a simplifica indexarea stive din cadrul programul utilizatorului (indexare pozitivă) și pentru a simplifica afișarea conținutului stivei de pe un panou frontal" [8]. Motivul pentru care procesoarele moderne (inclusiv gama x86-64 a procesoarelor pe 64 biți) au încă o stivă cu creștere „în jos" este acela că acestea utilizează aceeași arhitectură x86 pentru a fi compatibile cu 8086.

Procesoarele x86-64 permite salvarea pe stivă doar a valorilor pe 16 sau 64 biți, și nu se pot salva pe stivă direct valori pe 8 sau 32 biți. Pentru a se salva pe stivă o variabilă de memorie pe 8 biți (ByteVar), de exemplu, poate fi utilizată o instrucțiune de forma:

```
push   word ptr ByteVar
```

Deși „salvarea" unei variabile pe 8 biți (BYTE) este relativ sigură cu instrucțiunea de mai sus, trebuie avut grijă atunci când valoarea va fi „descărcată" de pe stiva într-o locație de memorie de 8 biți. Salvând o variabilă de 8 biți (ByteVar) cu instrucțiunea de mai sus se vor salva pe stivă doi octeți, octetul din variabila ByteVar și octetul imediat următor din memorie. Programul poate ignora pur și simplu octetul suplimentar pe care îl introduce pe stivă. Refacerea unei astfel de valori cu POP nu este chiar atât de simplă. În general, nu este rău dacă depuneți acești doi octeți pe stivă. Cu toate acestea, poate fi un dezastru dacă refaceți valoarea pe 16 biți și ștergeți (suprascrieți) și următorul octet din memorie. Există doar două soluții pentru această problemă. În primul rând, se poate reface valoarea de 16 biți într-un registru (precum AX) și apoi se va stoca partea mai nesemnificativă a registrului (AL) în variabila pe 8 biți. A doua soluție este să se rezerve un octet suplimentar pentru variabila pe 8 biți pentru a reține întregul cuvânt care va putea fi refăcut cu un POP. Majoritatea programelor folosesc ce-a de a doua abordare.

Instrucțiunile PUSH și POP nu modifică indicatorii de condiție (*flag*-urile).

4.2.6. Instrucțiunile PUSHF(Q) și POPF(Q)

Perechile de instrucțiuni PUSHF și POPF, respectiv PUSHFQ și POPFQ permit salvarea sau restaurarea registrului de stare al procesorului pe 16 respectiv 64 biți (FLAGS sau RFLAGS). Aceste două instrucțiuni oferă un mecanism pentru modificarea biților de stare ai procesorului.

În modul pe 64 biți, operația implicită a instrucțiunii este de a decrementa indicatorul

stivei (RSP) cu 8 și depune RFLAGS pe stivă. Funcționarea pe 16 biți este acceptată și decrementează RSP cu 2. Nu există în modul pe 64 biți o codare pentru operandul pe 32 biți (EFLAGS). Când se depune conținutul RFLAGS pe stivă, indicatorii de condiție VM și RF (biții 16 și 17) nu sunt copiați; în schimb, valorile pentru acești indicatori sunt șterse în imaginea RFLAGS stocată pe stivă.

Funcționarea instrucțiunilor PUSHF(Q) și POPF(Q) poate fi descrisă de algoritmii:

```
pushf
        RSP = RSP - 2
        [RSP] =  FLAGS
popf
        FLAGS = [RSP]
        RSP = RSP + 2
pushfq
        RSP = RSP - 8
        [RSP] =  RFLAGS
popfq
        RFLAGS = [RSP]
        RSP = RSP + 8
```

4.3. Instrucțiuni de conversie a datelor.

Instrucțiunile de conversie a datelor efectuează diverse transformări de date, cum ar fi dublarea dimensiunii operandului prin extensia semnului, conversia între formatele *little-endian* în *big-endian*.

4.3.1. Conversii de tip cu extinderea de semn

Instrucțiunile de conversie de tip convertesc octeți în cuvinte, cuvinte în dublu-cuvinte și dublu-cuvinte în cvadruplu-cuvinte. Aceste instrucțiuni sunt utile în special pentru conversia numerelor întregi în formate întregi mai mari, deoarece efectuează extensii de semn (a se vedea Figura 4.4).

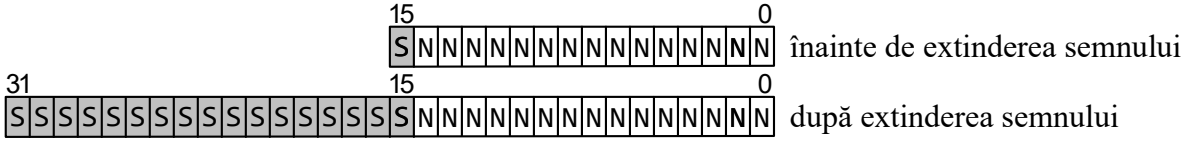

Figura 4.4: Extinderea de semn pt un număr de 16 biți la 32 biți.

Sunt două tipuri de instrucțiuni de conversie de tip: doar simplă conversie sau transfer și conversie.

Instrucțiunile de conversie simplă sunt: The CBW (Convert Byte to Word), CWDE (Convert Word to Doubleword Extended), CDQE (Convert Doubleword to Quadword Extended), CWD (Convert Word to Doubleword), and CDQ (Convert Doubleword to Quadword), CQO (Convert Quadword to Octaword) realizează extensia semnului pentru o dimensiune dublă a operandului sursă. Aceste instrucțiuni nu au operanzi.

```
cbw             ; AX ← SignExtend(AL)
cwde            ; EAX ← SignExtend(AX)
cdqe            ; RAX ← SignExtend(EAX)
cwd             ; DX:AX ← SignExtend(AX)
cdq             ; EDX:EAX ← SignExtend(EAX)
```

```
        cqo             ; RDX:RAX ← SignExtend(RAX)
```

Instrucțiunea CBW copiază semnul (bitul 7) al octetului din registrul AL în fiecare poziție a octetului superior al registrului AX (AH). Instrucțiunea CWDE copiază semnul (bitul 15) al cuvântului din registrul AX în fiecare poziție biți a cuvântului superior al registrului EAX.

Instrucțiunea CWD copiază semnul (bitul 15) al cuvântului din registrul AX în fiecare bit din registrul DX. Instrucțiunea CDQ copiază semnul (bitul 31) al dublu-cuvântului din registrul EAX în fiecare bit din registrul EDX. Instrucțiunea CQO copiază semnul (bitul 63) al cvadruplu-cuvântului din registrul RAX în fiecare bit din registrul RDX. Instrucțiunea CWD poate fi utilizată pentru a extinde cu semn deîmpărțitul de la cuvântul din registrul AX în perechea de regiștri DX:AX folosiți de o instrucțiune de împărțire cu semn (IDIV) de cuvinte. Similar instrucțiunea CDQ poate fi utilizată pentru a extinde deîmpărțitul (dublu-cuvânt) din EAX în EDX:EAX, instrucțiunea CQO poate fi utilizată pentru a pentru a extinde deîmpărțitul (cvadruplu-cuvânt) din RAX în RDX:RAX.

Dacă se dorește extinderea unei valori de opt biți la 32 sau 64 biți se pot utiliza aceste instrucțiuni în secvență, precum în următoarele exemple:

```
; Extinderea semnului din AL în DX:AX
        cbw             ; AX ← SignExtend(AL)
        cwd             ; DX:AX ← SignExtend(AX)
; Extinderea semnului din AL în EAX
        cbw             ; AX ← SignExtend(AL)
        cwde            ; EAX ← SignExtend(AX)
; Extinderea semnului din AL în RDX:RAX
        cbw             ; AX ← SignExtend(AL)
        cwde            ; EAX ← SignExtend(AX)
        cdqe            ; RAX ← SignExtend(EAX)
        cqo             ; RDX:RAX ← SignExtend(RAX)
```

Se poate utiliza și instrucțiunea MOVSX (Move with Sign-Extension) pentru a realiza extensii de semn de la 8 la 16, 32 sau 64 biți.

Instrucțiunea MOVSX este o formă generalizată a instrucțiunilor CBW, CWDE sau CDQE. Aceasta va extinde cu semn o valoare pe 8 biți la una pe 16, 32 sau 64, sau o valoare de 16 biți la 32 sau 64. Există și o variantă a instrucțiunii pe 64 biți: MOVSXD, care poate extinde cu semn o valoare pe 32 biți la una pe 64 biți. Forme admise pentru această instrucțiune sunt:

```
        movsx  reg16, reg8/mem8       ; Move BYTE to WORD with sign-extension
        movsx  reg32, reg8/mem8       ; Move BYTE to DWORD with sign-extension
        movsx  reg32, reg16/mem16     ; Move WORD to DWORD, with sign-extension
        movsxd reg16, reg16/mem16     ; Move WORD to WORD with sign-extension
        movsxd reg32, reg32/mem32     ; Move DWORD to DWORD, with sign-extension
        movsxd reg64, reg32/mem32     ; Move DWORD to QWORD, with sign-extension
```

Se poate observa că tot ce se poate face cu instrucțiunile CBW și CWDE, se poate realiza cu instrucțiuni MOVSX(D):

```
        movsx  ax, al               ; CBW
        movsx  eax, ax              ; CWDE
        movsx  eax, al              ; CBW urmata de CWDE
```

Cu toate acestea, instrucțiunile CBW, CWDE și CDQE sunt mai scurte și uneori mai rapide. Nu există echivalențe directe de MOVSX(D) pentru instrucțiunile CWD, CDQ și CQO.

Instrucțiunea MOVZX funcționează la fel ca instrucțiunea MOVSX, cu excepția faptului că

extinde valorile fără semn prin extensie cu zero în loc de valori cu semn prin extensie de semn. Sintaxa este aceeași ca și pentru instrucțiunile MOVSX, cu excepția, desigur, că se utilizează mnemonica MOVZX în loc de MOVSX. Forme admise pentru această instrucțiune sunt:

```
movzx  reg16, reg8/mem8       ; Move BYTE to WORD, zero-extension
movzx  reg32, reg8/mem8       ; Move BYTE to DWORD, zero-extension
movzx  reg64, reg8/mem8       ; Move BYTE to QWORD, zero-extension
movzx  reg32, reg16/mem16     ; Move WORD to DWORD, zero-extension
movzx  reg64, reg16/mem16     ; Move WORD to QWORD, zero-extension
```

Dacă se dorește să extindeți cu zero un registru de 8 biți la 16 biți (de exemplu, din AL în AX), o instrucțiune MOV simplă este mai rapidă și mai scurtă decât MOVZX. De exemplu,

```
mov    bh, 0
```

este mai rapidă și mai scurtă decât:

```
movzx  bx, bl
```

Desigur dacă se transferă valoarea unui alt registru pe 16 biți, de exemplu,

```
movzx  bx, al
```

instrucțiunea MOVZX este mai eficientă.

4.3.2. Instrucțiunea BSWAP

Instrucțiunea BSWAP (Byte Swap), disponibilă, convertește între valorile *little-endian* și *big-endian* pe 32 sau 64 biți. Această instrucțiune acceptă doar un singur operand de tip registru de uz general pe 32 sau 64 biți. Acesta schimbă ordinea octeților constituenți ai unui număr (a se vedea Figura 4.5).

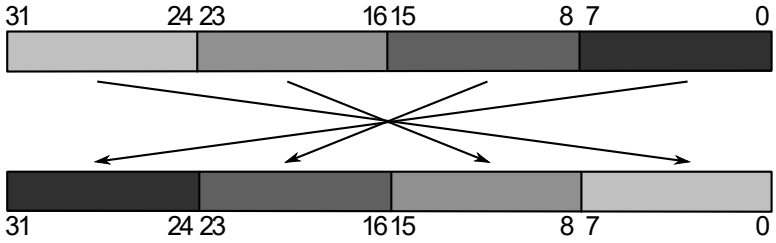

Figura 4.5. BSWAP pentru un număr pe 32 biți.

Sintaxa instrucțiunii BSWAP este:

```
bswap  reg32/64
```

Familiile de procesoare Intel utilizează o organizare a memoriei cunoscută sub denumirea *little-endian*. În organizarea de tip *little-endian* a octeților, octetul cel mai puțin semnificativ al unei secvențe cu mai mulți octeți este stocată la cea mai mică adresă din memorie. De exemplu, biții 0 până la 7 dintr-o valoare de 32 biți apar la adresa cea mai mică; biții 8÷15 se găsesc la a doua adresă în memorie; biții 16÷23 apar în al treilea octet iar biții 24÷31 apar în al patrulea octet.

O altă organizație de memorie populară este *big-endian*. În schema *big-endian*, biții cei mai semnificativi (biții 24÷31) apar în prima adresă (cea mai mică), biții 16÷23 apar în cel de-al doilea octet, biții 8÷15 apar în cel de-al treilea octet iar biții 0÷7 apar în al patrulea octet (adresa cea mai mare).

Instrucțiunii BSWAP poate oferi acces la un „al doilea set” de regiștri de uz general de 32 biți, sau patru seturi de regiștri pe 16 biți. Dacă sunt utilizați doar regiștri de 32 biți, se poate

dubla numărul de regiștri disponibile folosind instrucțiunea BSWAP pentru a schimba datele într-un registru de 32 biți cu partea superioară a unui registru pe 64 biți. De exemplu, se pot păstra două valori de 32 biți în RAX și muta valoarea corespunzătoare în EAX. Similar se pot păstra patru valori de 16 biți într-un registru pe 64 biți.

Notă: BSWAP acceptă ca parametru doar regiștri de uz general pe 32 sau 64 biți. Pentru a converti valorile *big-endian* de 16 biți în valori *little-endian* de 16 biți, trebuie utilizată instrucțiunea XCHG. De exemplu, dacă AX conține o valoare *big-endian* de 16 biți, aceasta poate fi convertită la o valoare *little-endian* de 16 biți (sau invers) folosind:

```
        xchg    al, ah
```

Instrucțiunea BSWAP nu afectează nici un *flag* (indicator de condiție) din registrul RFLAGS.

4.4. Instrucțiuni aritmetice

Instrucțiunile aritmetice (binare) funcționează cu date numerice de 8, 16, 32 și 64 biți codificate ca numere întregi binare cu sau fără semn. Instrucțiunile aritmetice binare pot fi, de asemenea, utilizate în algoritmi care operează cu valori zecimale (BCD).

Procesoarele x86-64 oferă numeroase operații aritmetice: adunarea, scăderea, negarea aritmetică (inversarea semnului), înmulțirea, împărțirea întreagă și compararea a două valori. Instrucțiunile care implementează aceste operațiuni sunt: ADD, ADC, INC, XADD, SUB, SBB, DEC, MUL, IMUL, DIV, IDIV, CMP, NEG, CMPXCHG,

4.4.1. Instrucțiunile ADD, ADC

Instrucțiunile ADD (Add) și ADC (Add with Carry) sunt instrucțiuni folosite pentru a realiza operația de adunare a numerelor întregi fără semn sau a numerelor întregi cu semn (reprezentate în complement față de 2). Instrucțiunile ADD și ADC au următoarea formă:

```
        add     dest, sursa
        adc     dest, sursa
```

Aceste instrucțiuni adaugă conținutul operandului sursă la operandul de destinație. Instrucțiunea ADD realizează calculul: dest = dest + sursa, în timp ce ADC calculează dest = dest + sursa + CF unde CF reprezintă valoarea din *flag*-ul carry. Prin urmare, dacă *flag*-ul *carry* este 0 înainte de execuție, ADC se comportă exact ca instrucțiunea ADD.

Sintaxa instrucțiunilor ADD și ADC este identică și este similară cu cea a instrucțiunii MOV. Ca și MOV, există formulare speciale pentru registrul AL/AX/EAX/RAX care sunt mai eficiente:

```
        add     AL/AX/EAX/RAX, immY                     ; Y = 8/16/32
        add     regX, immY
        add     memX, immY
        add     regX, regX
        add     memX, regX
        add     regX, memX                              ; X = 8/16/32/64
    ; Formele pentru instrucțiunea ADC sunt identice cu cele ale instrucțiunii ADD.
```

Din formele de mai sus se poate observa că valorile imediate pot fi codate doar până la 32 biți. În cazul în care operandul destinație este pe 64 biți valoarea imediată va fi extinsă la valoarea (cu semn) pe 64 biți echivalentă prin extindere de semn.

Ambele instrucțiuni afectează identic *flag*-urile (indicatorii de condiție). Acestea modifică *flag*-urile după cum urmează:

- *flag*-ul *overflow* (OF) indică o depășire în aritmetică cu semn;

- *flag*-ul *carry* (CF) indică o depășire în aritmetică fără semn;
- *flag*-ul *sign* (SF), de semn indică un rezultat negativ (adică, dacă bitul cel mai nesemnificativ al rezultatului este 1);
- *flag*-ul *zero* (ZF) este setat (1) dacă rezultatul este zero;
- *flag*-ul *auxiliary carry* (AF), de transport auxiliar conține 1 dacă este detectată depășire pentru cifra BCD cea mai nesemnificativă din rezultat în reprezentare BCD;
- *flag*-ul *parity* (PF) este setat sau șters în funcție de valoarea celor mai nesemnificativi opt biți ai rezultatului. Dacă există un număr egal de biți de 1 în cei 8 biți ai rezultatului, instrucțiunile ADD/ADC vor seta indicatorul de paritate pe unul (pentru a indica paritate pară); dacă număr de biți din rezultat este impar, instrucțiunile ADD/ADC șterg *falg*-ul de paritate (pentru a indica paritate impară);
- instrucțiunile ADD/ADC nu afectează alte *flag*-uri.

Instrucțiunile de adunare ADD și ADC permit operanzi pe 8, 16, 32 sau 64 biți. Atât operanzii sursă cât și cei destinate trebuie să aibă aceeași dimensiune.

Deoarece nu există posibilitatea de a aduna două locații de memorie, operanzii din memorie trebuie încărcați în regiștri dacă se dorește obținerea sumei a două variabile. Următoarele exemple de cod demonstrează forme posibile pentru instrucțiuni de adunare:

```
; S = X + Y
        mov     rax, X
        add     rax, Y
        mov     S, rax
```

Dacă se dorește obținerea sumei mai multor valori, aceasta se poate calcula cu ușurință într-un singur registru:

```
; S = X + Y + Z + T
        mov     rax, X
        add     rax, Y
        add     rax, Z
        add     rax, T
        mov     S, rax
```

Un lucru pe care programatorii începători îl uită adesea este faptul că este posibilă adunarea unui registru la o locație de memorie. Uneori, programatorii chiar cred că ambii operanzi trebuie să fie în regiștri. Procesorul x86-64 este un procesor care permite utilizarea modurilor de adresare a memoriei cu diverse instrucțiuni precum ADD. Adesea este mai eficient să se profite de capacitățile de adresare a memoriei.

```
; Y = X + Y
        mov     rax, X              ; rezultatul este corect
        add     Y, rax              ; deoarece adunarea este comutativa
```

Deseori, începătorii implementează operația de adunare ineficient după modelul secvențelor de mai jos:

```
; o solutie total ineficienta
        mov     rax, X
        mov     rcx, Y
        add     rax, rcx
        mov     Y, rax
; o solutie mai buna dar inca ineficienta
```

```
        mov     rax, Y
        add     rax, X
        mov     Y, rax
```

Desigur, dacă se dorește adunarea unei constante la o locație de memorie, este necesară o singură instrucțiune. Procesorul x86-64 vă permite să adăugați direct o constantă la o locație de memorie:

```
; var = var + 7
        add     var, 7
```

Există forme speciale ale instrucțiunilor `ADD` și `ADC` care adaugă o constantă la registrul `AL`, `AX`, `EAX` sau `RAX`. Aceste forme sunt mai scurte cu un octet decât adunarea unei constante la un alt registru. Există și instrucțiuni care oferă, de asemenea, forme mai scurte atunci când utilizați registrul acumulator; prin urmare, ar trebui, pe cât posibil, să se păstreze valorile pentru calcule în registrul acumulator (`AL`, `AX`, `EAX` sau `RAX`).

O altă optimizare vizează utilizarea constantelor cu semn în instrucțiunile `ADD` și `ADC`. Dacă o valoare este în intervalul `-128` ÷ `127`, instrucțiunile `ADD` și `ADC` vor extinde cu semn constanta imediată de 8 biți până la dimensiunea de destinație necesară (16, 32 sau 64). Prin urmare, ar trebui utilizate constante mici, dacă este posibil, cu instrucțiunile `ADD` și `ADC`.

4.4.2. Instrucțiunea INC

Instrucțiunea `INC` (Increment by 1) adaugă 1 la operandul destinație. Cu excepția *flag*-ului *carry* (`CF`), `INC` setează *flag*-urile la fel precum un apel „`add dest, 1`”.

Operandul destinație al instrucțiunii `INC` poate fi un registru sau o locație de memorie de 8, 16, 32 sau 64 biți. Instrucțiunea are două forme:

```
        inc     regx
        inc     memx                                    ; X = 8/16/32/64
```

Instrucțiunea `INC` este mai compactă dar nu neapărat mai rapidă decât „`add reg, 1`” sau „`add mem, 1`”. La procesoarele moderne, `ADD` nu este mai lentă decât `INC`, dar, de obicei, nu este nici mai rapidă, deci se preferă `INC` pentru motive de dimensiune a codului.

Instrucțiunea `INC` este foarte importantă deoarece adăugarea valorii `1` într-un registru este o operație foarte frecventă. Incrementarea contoarelor pentru buclele repetitive sau a indecșilor vectorilor este o operație des întâlnită, perfectă pentru instrucțiunile `INC`. Faptul că `INC` nu afectează *flag*-ul *carry* (`CF`) este foarte important. Această instrucțiune permite actualizarea unui contor pentru bucle fără a modifica *flag*-ul *carry*, pe când utilizarea unei instrucțiuni `ADD` cu un operand imediat de valoare 1 pentru a efectua o operație de incrementare va actualiza *flag*-ul CF.

4.4.3. Instrucțiunea XADD

Instrucțiunea `XADD` (Exchange and Add) interschimbă primul operand (operandul destinație) cu cel de-al doilea operand (operandul sursă), apoi încarcă suma celor două valori în operandul destinație. Operandul destinație poate fi un registru sau o locație de memorie; operandul sursă este un registru. Instrucțiunea are formele:

```
        xadd    regx, regx
        xadd    memx, regx                              ; X = 8/16/32/64
```

Apelul unei instrucțiuni `XADD` este echivalent cu secvența de instrucțiuni:

```
        xchg    dest, sursa   ; echivalent cu:
```

```
        add     dest, sursa   ; xadd dest, sursa
```

Atât la un apel al instrucțiunii XADD ca și la un apel ADD, în operandul destinație se va găsi suma celor două numere (dest=dest+sursa). Diferența între XADD și ADD constă în valoarea care se va afla în operandul sursă: după instrucțiunea XADD operatorul destinație va conține valoarea inițială a operandului sursă și nu își păstrează valoarea sa inițială ca în cazul instrucțiunii ADD.

XADD modifică *flag*-urile la fel ca și instrucțiunea ADD. Instrucțiunea XADD acceptă operanzi pe 8, 16, 32 sau 64 biți. Ambii operanzi trebuie să aibă aceeași dimensiune.

4.4.4. Instrucțiunile SUB și SBB

Instrucțiunile SUB (Subtract) și SBB (Subtract with Borrow) efectuează scăderi de numere întregi. Acestea evaluează rezultatul pentru operanzi întregi semn sau fără semn și setează *flag*-urile OF și CF pentru a indica o depășire în rezultatul cu respectiv fără semn.

Sintaxa lor este foarte similară cu cea a instrucțiunii de ADD:

```
        sub     dest, sursa
        sbb     dest, sursa
```

Instrucțiunea SUB calculează valoarea dest = dest – sursa, iar instrucțiunea SBB calculează dest = dest – sursa – CF. Este de reținut că operația de scădere nu este comutativă. Dacă se dorește calculul pentru dest = sursa – dest, vor trebui utilizate mai multe instrucțiuni (presupunând că operandul sursă trebuie păstrat).

Sintaxa instrucțiunilor SUB și SBB este identică și este similară cu cea a instrucțiunii ADD.

```
        sub     AL/AX/EAX/RAX, immY                     ; Y = 8/16/32
        sub     regX, immY
        sub     memX, immY
        sub     regX, regX
        sub     memX, regX
        sub     regX, memX                              ; X = 8/16/32/64
    ; Formele pentru instrucțiunea SBB sunt identice cu cele ale instrucțiunii SUB.
```

Ca și instrucțiunile ADD/ADC, SUB/SBB acceptă valori imediate pe cel mult 32 biți. Dacă operandul destinație este pe 64 biți valoarea imediată este extinsă cu semn la 64 biți.

Instrucțiunile SUB, SBB[1] afectează *flag*-urile după cum urmează:

- *flag*-ul *zero* (ZF) este setat dacă rezultatul este zero. Aceasta se întâmplă numai dacă operanzii sunt egali pentru SUB și SBB;
- *flag*-ul *sign* (SF) specifică dacă rezultatul este negativ;
- aceste instrucțiuni setează *flag*-ul *overflow* dacă a apărut depășire la operația cu semn. (sau în engl. *Underflow*);
- setează *flag*-ul *auxiliary carry* (AF), după caz, pentru aritmetica BCD/ASCII;
- setează *flag*-ul *parity* (PF) în funcție de numărul biți care apar în valoarea celor mai semnificativi 8 biți ai rezultatului;
- instrucțiunile SUB și SBB setează *carry flag* (CF) dacă apare o depășire în cazul reprezentării fără semn.

Instrucțiunile SUB și SBB furnizează forme scurte pentru a scădea o constantă din registrul acumulator (AL, AX, EAX sau RAX). Din acest motiv, ar fi de preferat ca operațiunile aritmetice să se realizeze pe cât posibil în registrul acumulator. Instrucțiunile SUB și SBB oferă, de asemenea, o

1 Instrucțiunea SBB afectează *flag*-urile într-o manieră similară cu SUB, doar că trebuie ținut cont de faptul că SBB calculează dest–sursa–CF

formă mai scurtă atunci când realizează scăderea cu constante din intervalul `-128` ÷ `127` dintr-o locație de memorie sau un registru. Instrucțiunea va extinde automat cu semn valoarea cu semn pe opt biți la dimensiunea necesară înainte de a efectua scăderea.

Instrucțiunea `SUB` poate fi utilizată sub forma „`sub reg, reg`” pentru a inițializa un registru cu `0` și a înlocui astfel o instrucțiune „`mov reg, 0`”. Paralelizarea instrucțiunilor poate fi îmbunătățită folosind instrucțiuni (`SUB` sau `XOR`) pentru a șterge conținutul unui registru [9]. Pe procesoarele moderne instrucțiunea „`sub reg, reg`” este mai rapidă și are o codare mai scurtă decât „`mov reg, 0`”. Procesoarele Intel moderne recunosc acest format al instrucțiunii SUB, „realizează” că rezultatul va fi zero și încarcă registrul destinație direct dintr-un registru fizic zero. Astfel nu mai este nevoie de execuția propriu-zisă a instrucțiunii (rezultatul va fi încărcat direct în destinație). Instrucțiunea `SUB` nu poate înlocui un `MOV` în cazul în care se dorește inițializarea cu zero a unui registru fără a modifica *flag*-urile setate anterior de o altă instrucțiune (`MOV` nu modifică nici un flag, iar `SUB` modifică `OF`, `SF`, `ZF`, `AF`, `PF` și `CF`). Același comportament se aplică și pentru instrucțiunea `XOR` (tratată în secțiunea 4.5.1 de la pagina 87).

4.4.5. Instrucțiunea DEC

Instrucțiunea `DEC` (Decrement by 1) scade `1` din operandul destinație. Cu excepția *flag*-ului *carry* (`CF`), `DEC` setează *flag*-urile la fel precum un apel „`sub dest, 1`”.

Operandul destinație al instrucțiunii `DEC` poate fi un registru sau o locație de memorie de 8, 16, 32 sau 64 biți. Instrucțiunea are două forme:

```
dec    regX
dec    memX                                    ; X = 8/16/32/64
```

Instrucțiunea `DEC` este similară cu instrucțiunea `INC` cu diferența că implementează funcția inversă (scădere în loc de adunare cu `1`).

4.4.6. Instrucțiunea CMP

Instrucțiunea `CMP` (Compare Two Operands) este identică cu instrucțiunea `SUB` cu o diferență crucială – nu stochează diferența în operandul destinație. Sintaxa pentru instrucțiunea `CMP` este foarte similară cu `SUB`, forma sa generică este:

```
cmp    dest, sursa
```

Forme specifice sunt:

```
cmp    AL/AX/EAX/RAX, immY                     ; Y = 8/16/32
cmp    regX, immY
cmp    memX, immY
cmp    regX, regX
cmp    regX, memX
cmp    memX, regX                              ; X = 8/16/32/64
```

Ca și la alte instrucțiuni se poate observa că valorile imediate pot fi codate doar pe 32 biți. În cazul în care primul operand este pe 64 biți valoarea imediată pe 32 biți va fi extinsă cu semn la 64 biți.

Instrucțiunea `CMP` actualizează indicatorii de condiție (*flag*-urile) în funcție de rezultatul operației de scădere (`dest - sursa`). Rezultatul comparației poate fi verificat prin testarea indicatoarelor de condiție corespunzătoare din registrul RFLAGS.

De obicei, după o instrucțiune `CMP`, este executată o instrucțiune de salt condiționată (a se vedea instrucțiunile `Jcc`). Acest proces în doi pași, compararea a două valori și setarea *flag*-

urilor, apoi testarea acestor *flag*-uri cu instrucțiunile de salt condiționat, este un mecanism foarte eficient pentru luarea deciziilor într-un program.

De exemplu dacă notăm operatorii instrucțiunii `CMP` după cum urmează:

```
cmp    op1, op2
```

instrucțiunea va efectua calculul `op1-op2` și stabilește *flag*-urile în funcție de rezultatul obținut. *Flag*-urile sunt setate în aceeași manieră ca instrucțiunea `SUB`, după cum urmează:

- `ZF`: *zero flag* este setat dacă și numai dacă `op1=op2`. Acesta este singura situație în care `op1-op2` produce un rezultat zero. Prin urmare, *flag*-ul *zero* este utilizat pentru a testa egalitatea sau neegalitatea;
- `SF`: *sign flag* este `1` dacă rezultatul este negativ. La prima vedere, s-ar putea crede că acest *flag* va fi setat dacă `op1` este mai mic decât `op2`, dar acest lucru nu este întotdeauna corect. De exemplu dacă `op1=7FFFh` și `op1=0FFFFh(-1)` scăderea produce valoarea 8000h, care este negativ (și, astfel, *sign flag*-ul va fi setat). Deci, pentru comparații cu semn, *sign flag*-ul oricum nu conține starea corespunzătoare. Pentru operanzi fără semn, dacă `op1=0FFFFh` și `op2=1`, `op1` este mai mare decât `op2`, dar diferența lor este `0FFFEh`, care este totuși negativ. După cum poate observa, *sign flag* și *overflow flag*, trebuie evaluate împreună, a compara două valori cu semn;
- `OF`: *overflow flag* este setat după o operațiune `CMP` dacă diferența de `op1` și `op2` produce o depășire (*overflow* sau *underflow*). Așa cum s-a putut observa mai sus, *sign flag* și *overflow flag* sunt ambele utilizate la efectuarea comparațiilor cu semn;
- `CF`: *carry flag* este setat după o operație `CMP` dacă scăderea `op2` din `op1` necesită un împrumut. Acest lucru se întâmplă numai atunci când `op1` este mai mic decât `op2` ca valori reprezentate fără semn;
- `PF`: este setat în funcție de numărul de biți de 1 dintre cei mai nesemnificativi 8 biți ai rezultatului;
- `AF`: indică dacă operația a produs un împrumut pentru cei mai nesemnificativi 4 biți (nibble) ai rezultatului.

Având în vedere că instrucțiunea `CMP` setează *flag*-urile în acest mod, comparația celor doi operanzi poate fi evaluată pe baza flag-urilor conform Tabelului 4.2:

Tabelul 4.2. Semnificația *flag*-urilor după o operație `CMP`.

Operații fără semn		*Operații cu semn*	
`ZF`	Egalitatea / Neegalitatea	`ZF`	Egalitatea / Neegalitatea
`CF`	`op1` < `op2` (`CF` = 1) `op1` ≥ `op2` (`CF` = 0)	`CF`	fără semnificație
`SF`	fără semnificație	`SF`	`op1` < `op2` (`SF` ≠ `OF`) `op1` ≥ `op2` (`SF` = `OF`)
`OF`	fără semnificație	`OF`	

4.4.7. Instrucțiunea CMPXCHG

Instrucțiunea `CMPXCHG` (Compare and Exchange) are următoarele forme:

```
cmpxchg regx, regx
cmpxchg memx, regx                                    ; X = 8/16/32/64
```

Operanzii trebuie să aibă aceeași dimensiune (8, 16, 32 sau 64 biți). Această instrucțiune folosește și registrul acumulator; alege automat AL, AX, EAX respectiv RAX pentru a se potrivi cu dimensiunea operanzilor.

Această instrucțiune compară registrul acumulator echivalent (AL, AX, EAX respectiv RAX) cu primul operand și setează *zero flag* (ZF) dacă sunt egali. Dacă da, atunci CMPXCHG copiază al doilea operand în primul. Dacă aceștia nu sunt egali, CMPXCHG copiază primul operand în acumulator. Următorul algoritm descrie această operație:

```
; cmpxchg op1, op2
if ({AL/AX/EAX/RAX} = operand1)
        ZF = 1      ; Seteaza Zero Flag
        op1 = op2
else
        ZF = 0      ; Resetează Zero Flag
        {AL/AX/EAX/RAX} = op1
endif
```

CMPXCHG este utilizat pentru managementul anumitor structuri de date ale sistemelor de operare pentru a realiza sincronizări între *task*-uri prin operații atomice[1] (MUTEX[2] sau semafoare). Desigur, dacă este posibilă încadrarea algoritmului de mai sus în codul dorit, instrucțiunea CMPXCHG poate fi folosită și în alte situații.

Observație: spre deosebire de instrucțiunea CMP, instrucțiunea CMPXCHG afectează doar *zero flag* (ZF). Nu pot fi testate alte flag-uri după un apel CMPXCHG, așa cum este posibil în urma uneui apel CMP.

4.4.8. Instrucțiunile CMPXCHG8B, CMPXCHG16B

Instrucțiunile CMPXCHG8B/CMPXCHG16B (Compare and Exchange Bytes) au următoarele forme:

```
cmpxchg8b mem64
cmpxchg16b mem128
```

Aceste instrucțiuni compara valoarea pe 64 biți în EDX:EAX (respectiv valoarea pe 128 biți în RDX:RAX dacă dimensiunea operandului este de 128 biți) cu operandul (destinație). Dacă valorile sunt egale, valoarea de 64 biți din ECX:EBX (respectiv valoarea pe 128 biți din RCX:RBX) este stocată în operandul destinație. În caz contrar, valoarea din operandul de destinație este încărcată în EDX:EAX (sau RDX:RAX). Operandul destinație este o locație de memorie cu 8 octeți (respectiv o locație de memorie de 16 octeți dacă dimensiunea operandului este de 128 biți). Pentru perechile de regiștri EDX:EAX și ECX:EBX, EDX și ECX conțin cei mai semnificativi 32 biți, iar EAX și EBX conțin cei mai nesemnificativi 32 biți pentru o valoare de 64 biți. Pentru perechile de înregistrări RDX:RAX și RCX:RBX, RDX și RCX conțin cei mai semnificativi 64 biți, iar RAX și RBX conțin cei mai nesemnificativi 64 biți de ordin redus, pentru o valoare de 128 biți.

Ca și CMPXCHG, aceste instrucțiuni modifică *flag*-ul *zero* (ZF) în funcție de rezultat. Nu afectează nici un alt *flag*.

4.4.9. Instrucțiunea NEG

1 O operație atomică reprezintă o operație care nu poate fi întreruptă. În programarea concurentă, o instrucțiune atomică înseamnă că nu se va efectua nici un schimb de context în timpul execuției acesteia, cu alte cuvinte nimic nu poate afecta execuția unei operații atomice.

2 MUTEX – din engl. MUTual EXclusion, este un mecanism de sincronizare pentru aplicarea limitării accesului la o resursă într-un mediu în care există mai multe fire de execuție.

Instrucțiunea NEG (Two's Complement Negation) înlocuiește valoarea operandului (operandul destinație) cu complementul acestuia față de doi. Această operație (negarea aritmetică) este echivalentă cu scăderea operandului din valoarea 0 (inversează efectiv semnul operandului). Operandul destinație poate fi situat într-un registru de uz general sau într-o locație de memorie pe 8, 16, 32 sau 64 biți.

Instrucțiunea NEG are următoarele forme de mai jos:

```
neg    regx
neg    memx                                    ; X = 8/16/32/64
```

Dacă operandul este zero, semnul său nu se schimbă, deși acest lucru resetează *flag*-ul *carry* (CF=0). Negarea oricărei alte valori setează *carry flag*. Negarea unui octet care conține valoarea -128, un cuvânt care conține -32768 sau un dublu-cuvânt care conține -2147483648 sau un cvadruplu-cuvânt care conține -9223372036854775808 (minimul reprezentabil cu semn) nu schimbă operandul, dar va seta *flag*-ul *overflow* (OF=1). Un apel al instrucțiunea NEG modifică *flag*-urile AF, SF, PF și ZF în același mod în care o face o instrucțiune SUB.

4.4.10. Instrucțiunea MUL

Instrucțiunea de înmulțire întreagă fără semn MUL (Unsigned Multiply) oferă primul impresie de neregularitate din setul de instrucțiuni din x86. Instrucțiuni precum ADD, ADC, SUB, SBB și multe altele din setul de instrucțiuni x86 utilizează doi operanzi. Din păcate, nu există suficienți biți în octetul opcode[1] al procesoarelor x86 pentru a susține toate instrucțiunile, astfel încât pentru instrucțiunile cu un singur parametru, de exemplu, INC, DEC și NEG, procesorul x86 folosește câmpul pentru parametru neutilizat ca extensie opcode. Acest lucru funcționează excelent pentru instrucțiunile cu un singur operand, permițându-le proiectanților Intel să codifice mai multe instrucțiuni (opt, de fapt) cu un singur opcode.

Chiar și așa, combinațiile opcode s-au dovedit insuficiente pentru a implementa diferite operații de înmulțire, astfel încât designerii de la Intel au proiectat instrucțiunile de înmulțite pentru a utiliza un singur operand. Câmpul pentru cel de-al doilea operand conține de fapt o extensie opcode, mai degrabă decât o valoare. Desigur, înmulțirea este o funcție cu doi operanzi, și procesorul x86 presupune întotdeauna că registrul acumulator (AL, AX, EAX sau RAX) este operandul destinație implicit. Această neregularitate face ca înmulțirea pe x86 să fie mai dificilă decât alte instrucțiuni, deoarece un operand trebuie să fie în acumulator. Intel a adoptat această abordare deoarece au considerat că programatorii vor folosi operația de înmulțire mult mai rar decât instrucțiuni precum ADD sau SUB.

Instrucțiunile de multiplicare MUL iau următoarele forme:

```
mul    regx
mul    memx                                    ; X = 8/16/32/64
```

Instrucțiunea MUL realizează o înmulțire fără semn a operandului destinație (deînmulțitul) – care este implicit situat în registrul AL, AX, EAX sau RAX (în funcție de mărimea operandului sursă) – și al doilea operand (operand sursă – înmulțitorul) și stochează rezultatul în operandul destinație. Operatorul sursă se află într-un registru de uz general sau într-o locație de memorie pe 8, 16, 32 sau 64 biți. Acțiunea acestei instrucțiuni și locația rezultatului depind de dimensiunea operandului (sursă), așa cum se arată în Tabelul 4.3.

Rezultatul este stocat în registrul AX, perechea de înregistrare DX:AX, perechea de înregistrare EDX:EAX sau perechea de înregistrare RDX:RAX (în funcție de dimensiunea

1 Opcode – abreviere din engl. operation code, este partea din instrucțiunea în limbaj mașină care specifică operația care trebuie efectuată.

operandului), jumătatea mai semnificativă a produsului fiind salvată în registrul AH, DX, EDX respectiv RDX. Dacă jumătatea mai semnificativă a produsului este 0, flag-urile *carry* și *overflow* sunt resetate (CF=0, OF=0); în caz contrar, acestea sunt setate (CF=1, OF=1).

Tabelul 4.3. Operanzii și destinația rezultatului instrucțiunii MUL.

Dimensiunea operandului	*Deînmulțitul*	*Înmulțitorul*	*Destinația (rezultatul)*
Byte	AL	reg_8/mem_8	AX
Word	AX	reg_{16}/mem_{16}	DX:AX
Dword	EAX	reg_{32}/mem_{32}	EDX:EAX
Qword	RAX	reg_{64}/mem_{64}	RDX:RAX

Instrucțiunea MUL înmulțește operanzi fără semn pe 8, 16, 32 sau 64 biți. Este de reținut că rezultatul a două valori reprezentate pe n biți necesită un spațiu de 2*n biți pentru a fi salvat corect. Prin urmare, dacă operandul este o cantitate de 8 biți, rezultatul va necesita 16 biți. De asemenea, un operand de 16, 32 sau 64 biți produce un rezultat pe 32, 64 respectiv 128. Acesta este motivul pentru care rezultatul este stocat într-o pereche de regiștri de aceeași dimensiune cu operandul instrucțiunii (AX=AH:AL, DX:AX, EDX:EAX respectiv RDX:RAX).

Instrucțiunea MUL corupe valorile flag-urilor AF, PF, SF și ZF care în urma unui apel al instrucțiunii vor avea valori nedefinite. În special, trebuie reținut faptul că, flagul de semn (SF) și flag-ul de zero (ZF) nu conțin valori semnificative după executarea acestei instrucțiuni.

4.4.11. Instrucțiunea IMUL

Instrucțiunea IMUL (Signed Multiply) operează pe operanzi cu semn. Deoarece Intel a încercat să generalizeze succesiv această instrucțiune pe diferite procesoare, există trei forme diferite ale acestei instrucțiuni în funcție de numărul operanzilor:

1. *Forma cu un operand* este identică cu cea utilizată de instrucțiunea MUL. Aici, operandul sursă (aflat într-un registru de uz general sau locație de memorie) se înmulțește cu valoarea din registrul AL, AX, EAX sau RAX (în funcție de dimensiunea operandului) și produsul (de două ori dimensiunea operandului sursă) este stocat în regiștrii AX, DX:AX, EDX:EAX, respectiv RDX:RAX (a se vedea Tabelul 4.3).

```
imul   regX
imul   memX                                  ; X = 8/16/32/64
```

2. La *forma cu doi operanzi*, operandul destinație (primul operand) este înmulțit cu operandul sursă (al doilea operand). Operandul de destinație poate fi un registru de uz general, iar operandul sursă poate fi o valoare imediată, un registru de uz general sau o locație de memorie. Produsul intermediar (de două ori mai mare decât operandul sursă) este trunchiat și stocat în operandul destinație.

```
imul   regX, regX
imul   regX, memX                            ; X = 16/32/64
imul   regX, immY                            ; Y = 8/16/32
```

3. *Forma cu trei operanzi* necesită un operand de destinație (primul operand) și doi operanzi sursă (cel de-al doilea și cel de-al treilea operand). Aici, primul operand sursă (care poate fi un registru de uz general sau o locație de memorie) este

înmulțit cu cel de-al doilea operand sursă (o valoare imediată). Produsul intermediar (de două ori mai mare decât primul operand sursă) este trunchiat și stocat în operandul destinație (un registru de uz general).

```
imul   regX, regX, immY                ; X = 16/32/64
imul   regX, memX, immY                ; Y = 8/16/32
```

O problemă a implementării formei inițiale a instrucțiunii (cea cu un singur operand) este aceea că nu poate accepta ca parametru o constantă; modul de adresare imediată fiind posibil doar pentru câmpul celui de-al doilea operand. Intel a descoperit rapid nevoia de a sprijini înmulțirea cu o constantă și a oferit un anumit suport pentru acest lucru în procesorul 286. Astfel a fost introdusă forma cu doi operanzi fapt deosebit de important pentru accesarea matricilor multidimensionale. Odată cu apariția procesorului 386, Intel a generalizat o formă a operației de înmulțire, permițând în final forma cu trei operanzi.

Instrucțiunile de forma „`imul reg, imm`” sunt de fapt o sintaxă specială pe care o oferă asamblorul. Codificările pentru aceste instrucțiuni sunt identice cu „`imul reg, reg, imm`”, unde asamblorul furnizează pur și simplu aceeași registru pentru ambii operanzi (primii doi operanzi).

Pe lângă numărul de operanzi, există mai multe diferențe între ultimele două forme și instrucțiunile cu un singur operand:

- Nu există o înmulțire pentru operanzii de 8 biți (operanzii cu valoare imediată pe 8 biți oferă pur și simplu o formă mai scurtă a instrucțiunii – procesorul extinde cu semn valoarea imediată la dimensiunea operandului destinație).
- Aceste instrucțiuni nu produc un rezultat de `2*n` biți (dublul dimensiunii operanzilor sursă). Adică, o multiplicare a unor numere pe `n` biți produce un rezultat trunchiat pe `n` biți. Aceste instrucțiuni setează *flag*-urile *carry* și *overflow* (`CF=1`, `OF=1`) dacă rezultatul nu se încadrează în registrul destinație.
- Spre deosebire de instrucțiunile `MUL`/`IMUL` standard (forma cu un singur operand), aceste versiuni (forme) permit un operand de tip valoare imediată.

Pentru formele cu doi operanzi, instrucțiunea `IMUL` este aproape la fel de generală ca instrucțiunea `ADD`, cu diferența că nu există posibilitatea înmulțirii regiștrilor pe 8 biți.

Cele trei forme ale instrucțiunii `IMUL` sunt similare, deoarece intern dimensiunea produsului este calculată la dublul dimensiunii operanzilor. Cu forma cu un singur parametru, produsul este stocat ca atare în destinație. La formele cu doi și trei operanzi, ambii operanzi (atât sursă cât și destinație) trebuie să aibă aceeași dimensiune, deci rezultatul intern este trunchiat la dimensiunea registrului destinație înainte de a fi stocat în acesta. Prin urmare, pentru a detecta o eventuală depășire, trebuie verificate *flag*-urile de *carry* (`CF`) sau *overflow* (`OF`). Dacă apare o depășire, jumătatea mai semnificativă a rezultatului se pierde.

Formele cu doi și trei operanzi pot fi, de asemenea, utilizate cu operanzi fără semn, deoarece jumătatea inferioară (mai puțin semnificativă) a produsului este aceeași, indiferent dacă operanzii sunt cu sau fără semn. Flag-urile `CF` și `OF` nu pot fi, însă, utilizate pentru a determina dacă jumătatea superioară a rezultatului este zero.

La fel ca și în cazul instrucțiunii MUL, *flag*-ul *zero* conține un rezultat nedeterminat după executarea unei instrucțiuni de înmulțire. *Flag*-ul *zero* (`ZF`) nu poate fi testat pentru a verifica dacă rezultatul este zero după o înmulțire. De asemenea, aceste instrucțiuni modifică și valoarea *sign flag*-ului (`SF`). Pentru a seta corect aceste *flag*-uri este necesară o comparație cu zero a rezultatului sau o testare a registrului destinație în care se găsește rezultatul.

4.4.12. Instrucțiunile DIV și IDIV

Instrucțiunile `DIV` și `IDIV` realizează o operație de împărțire fără respectiv cu semn a

valorii din regiștrii AX, DX:AX, EDX:EAX sau RDX:RAX (deîmpărțitul) cu operatorul sursă (împărțitorul) și stochează rezultatul în AX (AH:AL), DX:AX, EDX:EAX sau RDX:RAX. Operatorul sursă poate fi un registru de uz general sau o locație de memorie pe 8, 16, 32 sau 64 biți. Acțiunea acestei instrucțiuni depinde de dimensiunea operandului (a se vedea tabelul 4.4).

Tabelul 4.4. Operanzii și destinația rezultatului instrucțiunilor DIV/IDIV

Dimensiunea operandului	*Deîmpărțitul*	*Împărțitorul*	*Rezultat*	
			Cât	*Rest*
Byte	AX	reg_8/mem_8	AL	AH
Word	DX:AX	reg_{16}/mem_{16}	AX	DX
Dword	EDX:EAX	reg_{32}/mem_{32}	EAX	EDX
Qword	RDX:RAX	reg_{64}/mem_{64}	RAX	RDX

Procesoru x86, nu permite împărțirea unei valoare de 8 biți la alta, de exemplu, dacă împărțitorul este o valoare pe 8 biți, deîmpărțitul trebuie să fie o valoare pe 16 biți (dublul dimensiunii). Dacă valoarea deîmpărțitului este reprezentată fără semn aceasta trebuie extinsă cu zero la 16 biți. Acest lucru se poate realiza încărcând deîmpărțitul în registrul acumulator AL, AX, EAX sau RAX (după caz) și apoi resetând la zero valoarea registrului AH, DX, EDX respectiv RDX. Abia atunci instrucțiunea DIV va produce rezultatul corect. Dacă valoarea nu este extinsă cu zero înainte de a executa un DIV, acesta va produce rezultate incorecte!

Instrucțiunea IDIV este utilizată atunci când valorile întregi sunt reprezentate cu semn, valorile trebuie extinse cu semn din AL la AX, AX la DX:AX, EAX în EDX:EAX sau din RAX în RDX:RAX înainte de executarea instrucțiunii. Pentru acest lucru, pot fi folosite instrucțiunile CBW, CWD, CDQ, CQO (tratate în secțiunea 4.3.1. Conversii de tip cu extinderea de semn de la pagina 73) sau MOVSX. Dacă partea mai semnificativă nu conține deja biții mai semnificativi, valoarea care se află în acumulator (AL, AX, EAX sau RAX) trebuie extinsă cu semn înainte de a efectua operația de împărțire cu semn (IDIV). Nerespectarea acestui lucru poate produce rezultate incorecte.

Instrucțiunile de împărțire la x86 pot produce excepții. În primul rând, desigur, se poate încerca împărțirea unei valari cu zero. Mai mult, câtul poate fi prea mare pentru a se încadra în registrul destinație. De exemplu, împărțirea numărului 8000h reprezentat pe 16 biți la numărul 2 pe 8 biți produce rezultatul 4000h cu un rest 0. 4000h nu se poate reprezenta pe 8 biți. Dacă se întâmplă acest lucru sau împărțitorul este 0, x86 va genera o excepție[1] #DE (Divide Error). Apariția acestei excepții va determina întreruperea programului, și din acest motiv trebuie acordată o atenție sporită valorilor selectate atunci când este utilizată operația de împărțire.

Indicatoarele de condiție (Flag-urile) *auxiliary carry* (AF), *carry* (CF), *overflow* (OF), *parity* (PF), *sign* (SF) și *zero* (ZF) sunt nedefinite după o operație de împărțire DIV/IDIV. Dacă apare o depășire sau se încearcă o împărțire la zero se va genera o excepție.

Spre deosebire de instrucțiunea de înmulțire IMUL, instrucțiunea IDIV nu a dobândit forme speciale. Deoarece majoritatea programelor folosesc împărțirea mult mai rar decât înmulțirea, designerii de la Intel nu s-au străduit să creeze instrucțiuni speciale pentru operațiunea de împărțire. Astfel nu există nici o modalitate de a realiza o împărțire la o valoare imediată. Valoarea imediată trebuie încărcată într-un registru sau o locație de memorie și împărțirea se va efectua apoi la registrul sau locația de memorie respectivă.

1 O excepție este o condiție specială întâlnită în timpul executării unui program care este neașteptată sau anormală, pe care programatorul trebuie să o anticipeze și să o trateze corespunzător în codul programului.

4.5. Instrucțiuni logice

Instrucțiunile logice `AND` (ȘI), `OR` (SAU), `XOR` (SAU exclusiv) și `NOT` (NU logic) execută operațiunile booleene standard după care acestea sunt numite. Instrucțiunile `AND`, `OR` și `XOR` necesită doi operanzi, iar instrucțiunea `NOT` funcționează pe un singur operand.

4.5.1. Instrucțiunile AND, OR și XOR

Instrucțiunile logice funcționează de nivel de bit. Pentru fiecare instrucțiune există versiuni care operează cu numere pe 8, 16, 32 sau 64 biți. Instrucțiunile `AND`, `OR` și `XOR` au următoarele forme:

```
        and     AL/AX/EAX/RAX, immY
        and     regX, immY
        and     memX, immY
        and     regX, regX
        and     memX, regX                              ; Y = 8/16/32
        add     regX, memX                              ; X = 8/16/32/64
; Formele pentru instrucțiunea OR sunt identice cu cele ale instructiunii AND.
; Formele pentru instrucțiunea XOR sunt identice cu cele ale instructiunii AND.
```

Instrucțiunea `AND`, `OR` și `XOR` efectuează operația logică pe biți corespunzătoare între operandul destinație (primul operand) și cel sursă (al doilea operand) și stochează rezultatul în operandul destinație. Operatorul sursă poate fi o valoare imediată, un registru sau o locație de memorie; operandul de destinație poate fi un registru sau o locație de memorie (două locații de memorie nu pot fi folosite într-o singură instrucțiune). Regiștrii și locațiile de memorie pot fi pe 8, 16, 32 sau 64 biți. Valorile imediate pot fi doar pe 8, 16 sau 32 biți. Valoarea imediată va fi extinsă *cu semn* la dimensiunea operandului destinație. Operanzii instrucțiunii trebuie să aibă aceeași dimensiune.

Instrucțiunea `AND` efectuează operația logică AND pe biți (ȘI logic). Fiecare bit al rezultatului este setat pe `1` dacă ambii biți corespunzători ai primului și celui de-al doilea operand sunt `1`; altfel, este setat pe `0`.

Instrucțiunea `OR` efectuează operația logică OR pe biți (SAU logic). Fiecare bit din rezultatul instrucțiunii `OR` este setat la `0` dacă ambii biți corespunzători ai primului și celui de-al doilea operand sunt `0`; în caz contrar, fiecare bit este setat la `1`.

Instrucțiunea `XOR` efectuează operația logică XOR pe biți (SAU exclusiv). Fiecare bit al rezultatului este `1` dacă biții corespunzători ai operanzilor sunt diferiți; fiecare bit este `0` dacă biții corespunzători sunt identici.

Aceste instrucțiuni afectează indicatorii de condiție (*flag*-urile) după cum urmează:

- șterg *flag*-ul *carry* (`CF=0`);
- șterg *flag*-ul *overflow* (`OF=0`);
- setează *flag*-ul *zero* dacă rezultatul este zero (`ZF=1`), altfel îl șterg (`ZF=0`);
- copiază cel mai semnificativ bit al rezultatului în *sign flag;*
- setează *parity flag* în funcție de paritatea (numărul de biți de `1`) din octetul mai nesemnificativ al rezultatului;
- aceasta corupe valoarea *flag*-ul *auxiliary carry* (`AF` = *nedefinit)*.

Testarea *flag*-ului *zero* după aceste instrucțiuni este deosebit de utilă. Instrucțiunea `AND` setează *zero flag* (`ZF=0`) dacă cei doi operanzi nu au ambii biți de `1` în aceleași pozițiile de biți (deoarece acest lucru ar produce un rezultat zero). De exemplu, dacă operandul sursă conținea un singur bit, atunci `ZF` va fi setat dacă bitul din destinație corespunzător este `0`, altfel acesta va fi `1`. Instrucțiunea `OR` va seta indicatorul zero doar dacă ambii operanzi conțin biți de `0`. Instrucțiunea

XOR va seta *flag*-ul *zero* numai dacă ambii operanzi sunt egali. Operația XOR va produce un rezultat zero dacă și numai dacă cei doi operanzi sunt egali. Mulți programatori folosesc în mod obișnuit acest fapt pentru a șterge conținutul unui registru cu o instrucțiune de forma:

```
xor    reg, reg
```

care este mai scurtă decât instrucțiunea echivalentă:

```
mov    reg, 0
```

Ca și instrucțiunile de adunare și scădere, instrucțiunile logice AND, OR și XOR oferă forme speciale care implică registrul acumulator și date imediate. Aceste formulare sunt mai scurte și uneori mai rapide decât formele generale „reg, imm”.

Aceste instrucțiuni pot fi utilizate pentru a seta sau reseta anumiți biți din operandul destinație. Acest proces este cunoscut sub numele de mascare a datelor. O mască este o valoare utilizată pentru a forța anumiți biți în cadrul unei valori la 0 sau 1. O mască afectează de obicei anumiți biți dintr-un operand (forțându-i la 0 sau 1) și lasă restul biților neafectați. Utilizarea adecvată a măștilor permite extragerea de biți dintr-o valoare, inserarea de biți într-o valoare și împachetarea sau despachetarea tipului de date împachetate.

În urma executării unei instrucțiuni AND, dacă un bit din operandul sursă este 0, bitul corespunzător din rezultat va fi întotdeauna 0; dacă bitul este 1, valoarea bitului din rezultat va conține valoarea inițială din operandul destinație. Această proprietate poate fi folosită pentru a forța selectiv anumiți biți dintr-o valoare la 0, fără a afecta restul biților.

În măsura în care instrucțiunea AND poate fi utilizată pentru a forța valoarea anumitor biți la 0, instrucțiunea OR poate fi utilizată pentru a forța biții selectați la 1. În urma executării unei instrucțiuni OR, dacă un bit din operandul sursă este 0, bitul corespunzător din rezultat rămâne neschimbat; dacă bitul este 1, valoarea bitului din rezultat va deveni 1.

Din aceleași considerente ca și instrucțiunea SUB, XOR poate fi utilizată sub forma „xor reg, reg” pentru a inițializa un registru cu 0 și a înlocui o instrucțiune „mov reg, 0”, exceptând cazul când *flag*-urile trebuie păstrate; instrucțiunile XOR/SUB modifică *flag*-urile, iar instrucțiunea MOV nu modifică nici unul dintre *flag*-uri (a se vedea secțiunea 4.4.4 de la pagina 79).

4.5.2. Instrucțiunea NOT

Instrucțiunea NOT (One's Complement Negation) efectuează o operație negare logică pe biți (fiecare bit de 1 este resetat la 0 și fiecare bit de 0 este setat la 1) din operandul destinație și stochează rezultatul în același operand destinație. Operandul destinație poate fi un registru sau o locație de memorie pe 8, 16, 32 sau 64 biți. Aceasta are formele:

```
not    regx
not    memx                                    ; X = 8/16/32/64
```

Instrucțiunea NOT nu modifică nici unui din indicatorii de condiție (*flag*-uri).

4.6. Instrucțiunile de deplasare

Procesoarele x86 acceptă trei instrucțiuni de deplasare: (SHL și SAL sunt aceeași instrucțiune): SHL (Shift Left), SAL (Shift Aritmetic Left), SHR (Shift Right) și SAR (Shift Aritmetic Right). Procesoarele 386 și generațiile ulterioare oferă două instrucțiuni suplimentare: SHLD (Double Precision Shift Left) și SHRD (Double Precision Shift Right).

Formele instrucțiunilor SHL/SAL, SHR, SAR sunt:

```
shl    regx, 1
```

```
        shl     memx, 1
        shl     regx, imm8
        shl     memx, imm8
        shl     regx, cl
        shl     memx, cl                                    ; X = 8/16/32/64
; SAL este un sinonim pentru SHL si foloseste aceleasi formate.
; SHR si SAR folosesc aceleasi formate ca si SHL.
```

Cel de-al doilea parametru al instrucțiunii poate fi constanta „1”, o valoare imediată pe 8 biți sau valoarea din registrul `CL`. Valoarea din `CL` sau a constantei imediate ar trebui să fie mai mică sau egală cu numărul de biți pe care este reprezentat operandul destinație. Numărul este mascat pe 6 biți. Instrucțiunile `SHL`/`SAL`, `SHR`, `SAR` acceptă operanzi destinație pe 8, 16, 32 sau 64 biți.

Există diferențe minore în modul în care aceste instrucțiuni de deplasare tratează *flag*-ul de *carry* atunci când operandul sursă (operandul al doilea) este diferit de valoarea `1`, dar de cele mai multe ori `CF` poate fi ignorat (în cazul în care se face o deplasare cu mai mult de o poziție).

4.6.1. Instrucțiunea SHL/SAL

Mnemonica `SHL` (Shift Left) și `SAL` (Shift Arithmetic Left) sunt sinonime. Acestea reprezintă aceeași instrucțiune și utilizează codări binare identice. Aceste instrucțiuni realizează deplasarea la stânga a fiecărui bit din operandul destinație de numărul de ori specificat de operandul al doilea (valoarea `1`, constanta pe 8 biți sau registrul `CL`). Locurile vacante din partea mai nesemnificativă sunt completate cu `0`, bitul cel mai semnificativ va fi încărcat în *carry flag* (a se vedea Figura 4.6).

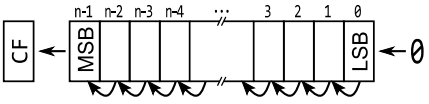

Figura 4.6. Instrucțiunea `SHL`/`SAL` cu o poziție, pentru un număr pe „n” biți.

Instrucțiunea `SHL`/`SAL` modifică indicatorii de condiție (*flag*-urile) după cum urmează:

- Dacă operandul al doilea (contorul) este zero, instrucțiunea `SHL`/`SAL` nu afectează nici un *flag*.
- *Flag*-ul *carry* conține cel mai semnificativ ultim bit deplasat din operand.
- *Flag*-ul *overflow* va conține `1` dacă în urma unei deplasări la stânga cu o singură poziție bitul cel mai semnificativ (bitul de semn) își schimbă valoarea. *Flag*-ul *overflow* nu este definit, dacă numărul de deplasări este mai mare ca `1`.
- *Flag*-ul *zero* va avea valoarea `1` dacă instrucțiunea produce un rezultat egal cu zero.
- *Flag*-ul *sign* va conține cel mai semnificativ bit din rezultat.
- *Flag*-ul *parity* va conține valoarea `1` dacă există un număr par de biți de `1` în octetul mai puțin semnificativ ai rezultatului.
- *Flag*-ul *auxiliary carry* este întotdeauna nedefinit după instrucțiunea `SHL`/`SAL`.

Deoarece deplasarea unei valori întregi la stânga este echivalentă cu înmulțirea acestei valori cu 2, instrucțiunea `SHL`/`SAL` poate fi utilizată pentru a efectua „înmulțiri” cu puteri ale lui 2.

```
        shl     rax, 1              ; echivalent cu RAX*2
        shl     rax, 2              ; echivalent cu RAX*4
        shl     rax, 3              ; echivalent cu RAX*8
        ; etc.
```

Instrucțiunea `SHL`/`SAL` înmulțește atât valori cu semn cât și fără semn, cu 2 pentru fiecare

deplasare cu o poziție. Această instrucțiune setează *flag*-ul *carry* dacă rezultatul nu se încadrează în operandul de destinație (adică se produce o depășire). De asemenea, această instrucțiune stabilește *flag*-ul *overflow* dacă rezultatul cu semn nu se încadrează în operatorul destinație. Aceasta se produce atunci când cel mai semnificativ bit al numărului își modifică valoarea.

4.6.2. Instrucțiunea SAR

Instrucțiunea `SAR` (Shift Arithmetic Right) deplasează biții operandului destinație la dreapta, reproducând în bitul cel mai semnificativ (bitul de semn) valoarea sa inițială (a se vedea Figura 4.7).

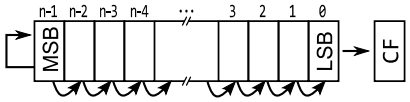

Figura 4.7. Instrucțiunea `SAR` cu o poziție pentru un număr pe „n” biți.

Instrucțiunea `SAR` modifică *flag*-urile după cum urmează:

- Dacă numărul de deplasări este zero, instrucțiunea `SAR` nu afectează nici un *flag*.
- *Flag*-ul *carry* conține ultimul bit deplasat din cel mai nesemnificativ bit al operandului.
- *Flag*-ul *overflow* va conține `0` dacă numărul de deplasări este `1`. O depășire nu poate apărea niciodată la această instrucțiune. Cu toate acestea, dacă numărul de deplasări nu este `1`, valoarea *flag*-ul *overflow* nu este definită.
- *Flag*-ul *zero* va fi `1` dacă deplasarea produce un rezultat egal cu zero.
- *Flag*-ul *sign* va conține cel mai semnificativ bit al rezultatului.
- *Flag*-ul *parity* va conține `1` dacă există un număr egal de biți de `1` în cei mai nesemnificativi 8 biți ai rezultatului.
- *Flag*-ul *auxiliary carry* este întotdeauna nedefinit după instrucțiunea `SAR`.

Scopul principal al instrucțiunii `SAR` este de a efectua o împărțire cu semn cu o putere a lui 2. Fiecare deplasare spre dreapta împarte valoarea cu 2. Deplasări multiple produc următoarele rezultate:

```
sar    rax, 1              ; impartire cu semn la 2
sar    rax, 2              ; impartire cu semn la 4
sar    rax, 3              ; impartire cu semn la 8
; etc.
```

Există o diferență foarte importantă între instrucțiunile `SAR` și `IDIV`. Instrucțiunea `IDIV` trunchiază întotdeauna spre zero în timp ce `SAR` trunchiază rezultatele spre rezultatul mai mic. Pentru rezultate pozitive, o deplasare aritmetică cu o poziție produce același rezultat ca o diviziune întreagă cu 2. Cu toate acestea, dacă câtul este negativ, `IDIV` trunchiază spre zero în timp ce `SAR` trunchiază spre -∞ (minus infinit). Următoarele exemple demonstrează diferența:

```
mov    eax, -7
cdq
mov    ecx, 2
idiv   ecx                 ; EAX = -3
mov    eax, -7
sar    eax, 1              ; EAX = -4
```

Trebuie ținut cont de acest lucru în cazul în care este utilizată instrucțiunea `SAR` pentru operațiuni de împărțire cu numere întregi.

4.6.3. Instrucțiunea SHR

Instrucțiunea SHR (Shift Right) realizează deplasarea biților din operandul de destinație spre dreapta completând locurile vacante cu zero (a se vedea Figura 4.8).

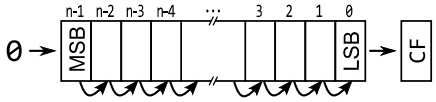

Figura 4.8. Operația instrucțiunii SAR cu o poziție pentru un număr pe „n” biți.

Instrucțiunea SHR afectează *flag*-urile după cum urmează:

- Dacă numărul de deplasări este zero, instrucțiunea SHR nu afectează nici un *flag*.
- *Flag*-ul *carry* (CF) conține cel mai nesemnificativ ultim bit deplasat din operand.
- Dacă numărul de deplasări este 1, *flag*-ul *overflow* (OF) va conține valoarea bitul cel mai semnificativ al valorii inițiale a operandului (cu alte cuvinte, această instrucțiune setează *flag*-ul de *overflow* (OF=1) dacă semnul operandului se schimbă). Cu toate acestea, dacă numărul de deplasări nu este 1, valoarea *flag*-ului *overflow* este nedefinită.
- *Flag*-ul *zero* va fi 1 dacă operațiunea produce un rezultat zero.
- *Flag*-ul *sign* va conține bitul cel mai semnificativ din rezultat, care este întotdeauna 0.
- *Flag*-ul *parity* va conține 1 dacă există un număr egal de biți de 1 în cei mai nesemnificativi 8 biți ai rezultatului.
- *Flag*-ul *auxiliary carry* este întotdeauna nedefinit după instrucțiunea SHR.

Instrucțiunea de deplasare la dreapta este utilă în special pentru despachetarea datelor. De exemplu, dacă se dorește extragerea (separarea) celor două caractere hexazecimale (*nibble*) din registrul AL, păstrând cifra mai semnificativă în AH, iar cifra mai puțin semnificativă în AL, acest lucru se poate realiza cu secvența de instrucțiuni:

```
mov    ah, al       ; copiaza octetul din AL in AH
shr    ah, 4        ; AH = cei mai semnificativi 4 biti
and    al, 0Fh      ; AL = cei mai nesemnificativi 4 biti
```

Deoarece deplasarea unei valori binare întregi fără semn cu o poziție la dreapta este echivalentă cu împărțirea respectivei valori la 2, instrucțiunea de SHR poate fi utilizată pentru a obține câtul împărțirii întregi fără semn la puteri ale lui 2 (nu se obține și restul).

```
shr    rax, 1              ; impartire fara semn la 2
shr    rax, 2              ; impartire fara semn la 4
shr    rax, 3              ; impartire fara semn la 8
; etc.
```

Împărțirea la doi folosind SHR funcționează numai pentru operanzi fără semn. Dacă, de exemplu, registrul RAX conține valoarea -1 și în urma executării instrucțiunii „shr rax, 1” rezultatul în RAX va fi (7FFFFFFFFFFFFFFFh), nu -1 sau 0 (valorile rotunjite sau trunchiate pentru -0.5). Pentru împărțirea unui număr întreg cu semn la o putere a lui 2, poate fi folosită instrucțiunea SAR.

4.6.4. Instrucțiunile SHLD și SHRD

Instrucțiunile SHLD (Double Precision Shift Left) și SHRD (Double Precision Shift Right) oferă operații de deplasare în dublă de precizie la stânga, respectiv la dreapta. Formele lor generice sunt:

```
shld   regx, regx, imm8
shld   memx, regx, imm8
shld   regx, regx, cl
shld   memx, regx, cl                                    ; X = 16/32/64
; SHRD foloseste aceleasi formate ca si SHLD.
```

Operandul destinație poate fi un registru sau o locație de memorie; operandul sursă este un registru. Primii doi operanzi trebuie să fie de aceeași dimensiune: 16, 32 sau 64 biți. Operandul de numărare este un număr întreg fără semn care poate fi stocat într-o valoare imediată pe 8 biți sau în registrul `CL`. Dacă valoarea operandului de numărare este mai mare decât dimensiunea operanzilor, rezultatul este nedefinit.

Instrucțiunile `SHLD` deplasează biții din operandul destinație spre stânga. Cei mai semnificativi biți sunt deplasați în *flag*-ul *carry* și cei mai semnificativi biți ai operandului sursă completează cei mai nesemnificativi biți din operandul destinație. Această instrucțiune nu modifică valoarea operandului sursă, ci folosește o copie temporară a acestuia în timpul deplasării. Operandul imediat specifică numărul de biți care trebuie deplasați. Dacă numărul este `n`, atunci `SHLD` mută bitul *n-1* în `CF`. De asemenea, cei mai semnificativi `n` biți din al doilea operand (operandul sursă) sunt copiați în cei mai nesemnificativi `n` biți ai primului operand (operandul destinație).

Instrucțiunea `SHLD` modifică indicatorii de condiție (*flag*-urile) după cum urmează:

- Dacă numărul de deplasări este zero, instrucțiunea `SHLD` nu afectează nici un *flag*.
- *Flag*-ul *carry* (`CF`) conține ultimul cel mai semnificativ bit deplasat din destinație.
- Dacă numărul de deplasări este `1`, *flag*-ul *overflow* (`OF`) va conține `1` dacă bitul de semn al operandului destinație se modifică în timpul deplasării. Dacă numărul de deplasări este mai mare ca `1`, `OF` este nedefinit.
- *Flag*-ul *zero* (`ZF`) va fi `1` dacă deplasarea produce un rezultat zero.
- *Flag*-ul *sign* (`SF`) va conține cel mai semnificativ bit din rezultat (bitul de semn).

Instrucțiunea `SHLD` este utilă pentru împachetarea datelor din mai multe surse diferite. De exemplu, să presupunem că se dorește crearea unui cuvânt compus din cei mai semnificativi 4 biți (high nibble / cifra hexazecimală) ai altor 4 cuvinte.

Puteți face acest lucru cu următorul cod:

```
mov    ax, Val4          ; incarcare valoarea a 4-a
shld   bx, ax, 4         ; copiere 4 biti din AX in BX
mov    ax, Val3          ; incarcare valoarea a 3-a
shld   bx, ax, 4         ; copiere 4 biti din AX in BX
mov    ax, Val2          ; incarcare valoarea a 2-a
shld   bx, ax, 4         ; copiere 4 biti din AX in BX
mov    ax, Val1          ; incarcare prima valoarea
shld   bx, ax, 4         ; BX contine cele 4 valori compuse.
```

Instrucțiunea `SHRD` este similară cu `SHLD`, cu excepția faptului că deplasează biții la dreapta și nu la stânga.

Instrucțiunea `SHRD` modifică valorile *flag*-urilor astfel:

- Dacă numărul de deplasări este zero, instrucțiunea `SHRD` nu afectează nici un *flag*.
- *Flag*-ul *carry* (`CF`) conține ultimul bit deplasat din bitul mai nesemnificativ al operandului destinație.
- Dacă numărul de deplasări este `1`, *flag*-ul *overflow* (`OF`) va conține `1` dacă bitul cel mai semnificativ al operandului se schimbă. Dacă numărul de deplasări este mai mare ca 1, `OF` este nedefinit.

- *Flag*-ul *zero* (ZF) va fi 1 dacă deplasarea produce un rezultat zero.
- *Flag*-ul *sign* (SF) va conține cel mai semnificativ bit din rezultat (bitul de semn).

4.7. Instrucțiuni de rotire a biților

Instrucțiunile de rotire deplasează biții, la fel ca și instrucțiunile de deplasare, cu diferența că biții deplasați din operand de către instrucțiunile de rotire sunt recirculați. Acestea includ ROL (Rotate Left), ROR (Rotate Right), RCL (Rotate Through Carry Left) și RCR (Rotate Through Carry Right). Aceste instrucțiuni iau formele:

```
        rol     regx, 1
        rol     memx, 1
        rol     regx, imm8
        rol     memx, imm8
        rol     regx, cl
        rol     memx, cl                                    ; X = 8/16/32/64
; ROR, RCL si RCR folosesc aceleasi formate ca si ROL.
```

Aceste instrucțiuni deplasează (rotesc) biții primului operand (operandul destinație) cu numărul de poziții specificate în al doilea operand și stochează rezultatul în operandul destinație. Operandul destinație poate fi un registru sau o locație de memorie pe 8, 16, 32 sau 64 biți; operandul de numărare este un număr întreg fără semn care poate fi valoare imediată sau o valoare în registrul CL. Numărul este mascat pe 6 biți.

4.7.1. Instrucțiunea ROL

Instrucțiunea ROL (Rotate Left) realizează deplasarea biților operandului destinație către poziții mai semnificative (stânga) cu un anumit număr de poziții, iar biții deplasați cei mai semnificativi sunt folosiți pentru a completa biții mai puțin semnificativi, vacanți (a se vedea Figura 4.9).

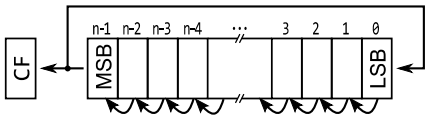

Figura 4.9: Operația de rotire la stânga cu o poziție pentru un număr pe „n” biți.

Dacă un număr pe n biți este rotit de n ori, ceea ce se va obține va fi valoarea inițială. Totuși, această operație va modifica unele *flag*-uri.

Instrucțiunea ROL modifică *flag*-urile după cum urmează:

- *Flag*-ul *carry* (CF) conține ultimul cel mai semnificativ bit deplasat din operand.
- Dacă deplasarea este cu o singură poziție, ROL setează *flag*-ul *overflow* (OF) dacă semnul se schimbă ca urmare a rotirii. Dacă deplasarea este cu mai multe poziții, *flag*-ul de *overflow* este nedefinit.
- Instrucțiunea ROL nu modifică *flag*-urile *zero* (ZF), sign (SF), parity (PF) sau auxiliary carry (AF).

Spre deosebire de instrucțiunile de *shift*-are, instrucțiunile de rotire nu modifică flag-urile ZF, SF, PF sau AF. Acest comportament face inutilă testarea acestor flag-uri după instrucțiunile de rotire. Dacă setarea acestor *flag*-uri este necesară pentru rezultatul instrucțiunii ROL, după testarea *flag*-urilor CF și OF (dacă este cazul), este necesară o comparație a rezultatului cu zero pentru a seta celelalte *flag*-uri. Acest lucru este valabil și pentru celelalte instrucțiuni de rotire.

4.7.2. Instrucțiunea ROR

Instrucțiunea ROR (Rotate Right) realizează deplasarea biților operandului destinație către poziții mai puțin semnificative (drepta) cu un anumit număr de poziții, iar biții deplasați cei mai nesemnificativi sunt folosiți pentru a completa biții mai semnificativi vacanți (a se vedea Figura 4.10).

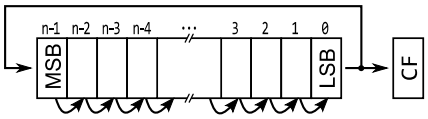

Figura 4.10. Operația de rotire la dreapta cu o poziție pentru un număr pe „n" biți.

Instrucțiunea ROR modifică *flag*-urile după cum urmează:

- *Flag*-ul *carry* (CF) conține ultimul cel mai nesemnificativ bit deplasat din operand.
- Dacă deplasarea este cu o singură poziție, ROR setează *flag*-ul *overflow* (OF) dacă semnul se schimbă ca urmare a rotirii. Dacă deplasarea este cu mai multe poziții, *flag*-ul de *overflow* este nedefinit.
- Instrucțiunea ROR nu modifică *flag*-urile *zero* (ZF), sign (SF), parity (PF) sau auxiliary carry (AF).

4.7.3. Instrucțiunea RCL

Instrucțiunea RCL (Rotate Through Carry Left), așa cum îi spune și numele, rotește biți spre stânga, prin *flag*-ul *carry* și înapoi în bitul zero (cel mai nesemnificativ) (a se vedea Figura 4.11).

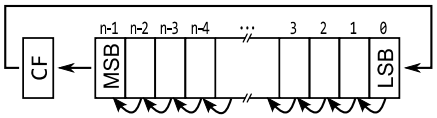

Figura 4.11. Operația instrucțiunii RCL cu o poziție pentru un număr pe „n" biți.

Dacă se rotește prin CF cu o poziție de n+1 ori, unde n este numărul de biți ai operandului, se va obține valoarea inițială. Totuși, valorile unor flag-uri vor fi modificate.

Instrucțiunea RCL modifică *flag*-urile după cum urmează:

- *Flag*-ul *carry* (CF) conține ultimul cel mai semnificativ bit deplasat din operand.
- Dacă deplasarea este cu o singură poziție, RCL setează *flag*-ul *overflow* (OF) dacă semnul se schimbă ca urmare a rotirii. Dacă deplasarea este cu mai multe poziții, steagul de preaplin este nedefinit.
- Instrucțiunea RCL nu modifică *flag*-urile *zero* (ZF), sign (SF), parity (PF) sau auxiliary carry (AF).

4.7.4. Instrucțiunea RCR

Instrucțiunea RCR (Rotate Through Carry Right) este complementul instrucțiunii RCL. Aceasta rotește biții mai nesemnificativi prin *flag*-ul *carry* și înapoi în biții cei mai semnificativi (a se vedea Figura 4.12).

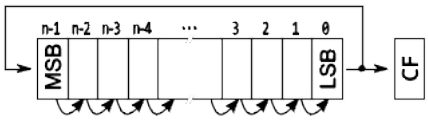

Figura 4.12. Operația de rotire la dreapta cu o poziție pentru un număr pe „n" biți.

Instrucțiunea RCR modifică *flag*-urile într-o manieră similară cu ROR:

- *Flag*-ul *carry* (CF) conține ultimul cel mai nesemnificativ bit deplasat din

operand.
- Dacă deplasarea este cu o singură poziție, `RCR` setează *flag*-ul *overflow* (`OF`) dacă semnul se schimbă ca urmare a rotirii. Dacă deplasarea este cu mai multe poziții, steagul de preaplin este nedefinit.
- Instrucțiunea `RCR` nu modifică *flag*-urile *zero* (`ZF`), *sign* (`SF`), *parity* (`PF`) sau *auxiliary carry* (`AF`).

4.8. Instrucțiuni pe biți

Operațiile pe biți sunt operațiunile care pot fi realizate cel mai ușor în limbajul de asamblare față de alte limbaje de programare. Aceasta deoarece, majoritatea limbajelor de programare de nivel înalt ascund programatorului reprezentarea internă a tipurilor de date. Instrucțiuni precum `AND`, `OR`, `XOR`, `NOT`, deplasările și rotirile permit indirect testarea, setarea, ștergerea, inversarea biților dintr-un număr. Chiar și limbajul de programare C++, renumit pentru operatorii săi de manipulare a biților, nu oferă toate capacitățile limbajului de asamblare.

Familia de procesoare x86, în special procesoarele 80386, merg mult mai departe. Pe lângă instrucțiunile logice standard și cele de deplasare și rotire, acestea oferă instrucțiuni pentru operarea cu biții din cadrul unui operand: testarea, setarea, ștergerea sau inversarea unor biți specifici dintr-un operand sau identificarea unei secvențe de biți. Aceste instrucțiuni sunt:

```
test   dest, sursa
bt     sursa, index
btc    sursa, index
btr    sursa, index
bts    sursa, index
bsf    dest, sursa
bsr    dest, sursa
```

Exceptând instrucțiunea `TEST`, instrucțiunile `BT`, `BTC`, `BTR`, `BTS`, `BSF` și `BSR` necesită operanzi de 16, 32, 64 biți (nu operează pe 8 biți).

4.8.1. Instrucțiunea TEST

Instrucțiunea `TEST` (Logical Compare) efectuează o operație logică ȘI (AND) între cei doi operanzi și modifică *flag*-urile în consecință, dar nu salvează rezultatul. Instrucțiunile `TEST` și `AND` împărtășește aceeași relație ca și instrucțiunile `CMP` și `SUB`.

Instrucțiunea TEST are următoarele forme:

```
test   regX, regX
test   memX, regX
test   regX, immY
test   memX, immY                                  ; Y = 8/16/32
test   rax/eax/ax/al, immY                         ; X = 8/16/32/64
```

Cei doi operanzi trebuie să aibă aceeași dimensiune. Valorile imediate pot fi reprezentate doar pe 8, 16 sau 32; în cazul când operandul destinație este pe 64 biți valoarea imediată pe 32 biți va fi extinsă cu semn la 64 biți.

Deoarece operația logică ȘI (AND) este comutativă iar instrucțiunea `TEST` nu modifică operandul destinație, asamblorul acceptă și forma de mai jos interschimbând automat ordinea operanzilor:

```
test   regX, memX     ; inlocuita cu: "test memX, regX" pt. X = 8/16/32/64
```

În mod obișnuit, această instrucțiune este utilizată pentru a vedea dacă un anumit bit conține valoarea 1.

```
test   al, 10000000b ; 10000000b = 80h = 128
```

Această instrucțiune va efectua un ȘI logic între valoarea din registrul AL și valoarea imediată 10000000b. Dacă bitul 7 (poziție pe care se află bitul de 1 din valoarea imediată) din AL conține 0, rezultatul este 0 și procesorul va reseta *flag*-ul *zero* (ZF=0). Dacă bitul 7 din AL conține 1, atunci rezultatul este diferit de 0 și instrucțiunea TEST setează *flag*-ul *zero* (ZF=1). *Zero flag* poate deci fi testat după această instrucțiune pentru a decide dacă AL conține 0 sau 1 în bitul testat (poziția pe care se află bitul de 1 din mască).

Instrucțiunea TEST poate fi utilizată pentru a verifica, de asemenea, dacă unul sau mai mulți biți dintr-un registru sau o locație de memorie sunt 0. De exemplu următoarea instrucțiune:

```
test   si, 0FF00h
```

Această instrucțiune va efectua un AND între valoarea din registrul SI și constanta 0xFF00. Aceasta va produce un rezultat non-zero (și, prin urmare, ZF=1) dacă cel puțin unul dintre biții 8÷15 este 1. Toți biții trebuie să fie zero pentru a seta *flag*-ul *zero* (ZF=0). Practic instrucțiunea de mai sus verifică dacă jumătatea superioară (octetul superior) este sau nu zero. Aceasta este utilă pentru regiștrii care nu au acces direct al doilea octet din componența lor (în exemplu SI), care spre deosebire de AX, BX, CX și DX nu pot verifica direct octetul superior prin intermediul regiștrilor AH, BH, CH sau DH.

Instrucțiunea TEST stabilește *flag*-urile identic cu instrucțiunea AND:

- șterg *flag*-ul *carry* (CF=0);
- șterg *flag*-ul *overflow* (OF=0);
- setează *flag*-ul *zero* dacă rezultatul este zero (ZF=1), altfel îl șterg (ZF=0);
- copiază cel mai semnificativ bit al rezultatului în *sign flag;*
- setează *parity flag* în funcție de paritatea (numărul de biți de 1) din octetul mai nesemnificativ al rezultatului;
- corupe valoarea *flag*-ul *auxiliary carry (*AF = *nedefinit)*.

Unii programatori utilizează instrucțiunea „`test reg, reg`” pentru a înlocui o comparație cu zero: „`cmp reg, 0`”. Cele două instrucțiuni sunt aproape identice cu excepția faptului că „`test reg, reg`” este mai scurtă decât „`cmp reg, 0`” cu un octet (exceptând cazul registrului AL când ambele codări au doi octeți), deoarece CMP necesită codarea în instrucțiune a valorii imediate 0 ca argument. Diferența este de doar un octet, deoarece valoarea imediată 0 este reprezentată pe 8 biți și va fi extinsă cu semn la dimensiunea registrului destinație (primul operand).

Diferența între cele două constă în faptul că, dacă instrucțiunea CMP este o operațiune aritmetică (realizează o scădere „`reg - 0`”, fără a salva rezultatul rezultatul), instrucțiunea TEST este o operație logică (execută un AND pe biți „`reg & reg`” și ignoră rezultatul), astfel încât se poate suspecta, în mod rezonabil, că acestea modifică *flag*-urile în mod diferit. Se dovedește însă, că ambele instrucțiuni modifică *flag*-urile într-o manieră aproape identică. Ambele instrucțiuni modifică biții OF, SF, ZF, AF, PF și CF din registrul FLAGS. Instrucțiunea TEST șterge întotdeauna OF și CF, dar același lucru îl face CMP când realizează comparația cu 0. Singura altă diferență este că instrucțiunea CMP va seta corect obscurul flag AF, în timp ce instrucțiunea de TEST lasă nedefinit conținutul acestuia. În cazul instrucțiunii „`cmp reg, 0`” *flag*-ul AF va fi întotdeauna resetat (AF=0) indiferent de valoarea registrului destinație, deci nu există nici o informație relevantă care se poate obține din *flag*-ul AF.

Prin urmare, se poate concluziona că nu există nici o situație în care „`test reg, reg`” va oferi ceva deosebit față de „`cmp reg, 0`”, și invers. Cele două instrucțiuni par a fi complet

echivalente, cu excepția unui octet diferență în lungimea codării instrucțiunilor.

Intel, *Assembly/Compiler Coding Rule 36* din *Architectures Optimization Reference Manual*, secțiunea 3.5.1.8, recomandă: „Utilizați instrucțiunea `TEST` în loc de `AND`, atunci când rezultatul operației ȘI logic nu este utilizat. Acest lucru economisește µops[1] în timpul executării. Utilizarea instrucțiunii `TEST` al unui registru cu sine însuși în locul unui `CMP` al registrului cu zero, elimină necesitatea codificării constantei zero și economisește spațiu de codificare. Evitați să comparați o constantă cu un operand de memorie. Este de preferat ca un operandul de memorie să fie încărcat într-un registru și ulterior acesta să fie comparat cu o constantă” [9].

4.8.2. Instrucțiunile BT, BTS, BTR și BTC

Aceste instrucțiunile `BT` (Bit Test), `BTS` (Bit Test and Set), `BTR` (Bit Test and Reset) și `BTC` (Bit Test and Complement) au următoarele forme:

```
        bt      regx,  regx
        bt      memx,  regx
        bt      regx, imm8
        bt      memx, imm8                              ; X = 16/32/64
; BTC, BTR, BTS au aceleași formate ca si BT.
```

Aceste instrucțiuni de testare și modificare a biților (a se vedea Tabelul 4.5) funcționează pe un singur bit, dintr-un registru sau locație de memorie pe 16, 32 sau 64 biți. Locația bitului din număr este specificat (în al doilea operand) ca un index (deplasament) față de bitul cel mai puțin semnificativ al operandului. Indexul utilizează practic numerotarea standard pentru biții unui număr: bitul `0` este bitul cel mai nesemnificativ, bitul `1` este cel al doilea bit, până la bitul `n-1` care este bitul cel mai semnificativ al unui număr pe `n` biți. Când procesorul identifică bitul care urmează a fi testat și modificat, mai întâi încarcă flag-ul `CF` cu valoarea anterioară a acestuia. Apoi atribuie o nouă valoare pentru bitul selectat, determinată de instrucțiunea utilizată (exceptând instrucțiunea `BT` care nu modifică bitul selectat).

Tabelul 4.5. Instrucțiunile de testare și modificare pe biți.

Instrucțiunea	*Valoarea flag-ului CF*	*Valoarea bitului selectat*
BT (Bit Test)	`CF` *flag* ← Bit selectat	Nemodificat
BTS (Bit Test and Set)		Bit ← `1`
BTR (Bit Test and Reset)		Bit ← `0`
BTC (Bit Test and Complement)		Bit ← NOT (Bit)

De exemplu, instrucțiunea:

```
        bt      si, 15
```

va copia bitul cel mai semnificativ, al 16-lea (bitul cu numărul 15) din registrul pe 16 biți `SI` în *flag*-ul de *carry* (`CF`).

Instrucțiunile `BT/BTS/BTR/BTC` operează doar pe operanzi de 16, 32 sau 64 biți. Faptul că această instrucțiune nu operează pe 8 biți nu constituie o limitare deoarece dacă se dorește testarea bitului unui octet, acești biți sunt aceiași cu primii 8 biți ai cuvântului din care octetul face parte. Astfel, dacă se dorește testarea unui bit al unui registru pe 8 biți, se pot testa biții de la `0` la `7` ai registrului extins pe 16 biți (`AX` pentru `AL`, `SI` pentru `SIL`, `R8W` pentru `R8B`, etc.).

1 µops – din engl. micro-operation (micro-operație) sunt operațiunile elementare interne procesorului pe baza cărora se execută intrucțiunile.

Aceste instrucțiuni utilizează de fapt restul împărțirii (modulo) valorii celui de-al doilea operand la 16, 32 sau 64 pentru a determina valoarea indexului bitului selectat, unde valorile de 16, 32 sau 64 reprezintă dimensiunea primului operand (`WORD`, `DWORD` resperctiv `QWORD`). Prin urmare, o instrucțiune „`bt si, 17`" va încărca în `flag`-ul `CF` valoarea bitului 1 (deoarece restul împărțirii lul `17` la la `16` este `1`).

După cum se poate observa și din Tabelul 4.5, toate aceste instrucțiuni copiază bitul selectat în *carry flag*. În plus față de simplul test al instrucțiunii `BT`, instrucțiunea `BTS` setează bitul selectat pe `1`, `BTR` îl resetează la valoarea `0`, iar `BTC` îi inversează valoarea, după copierea valorii inițiale în CF.

Instrucțiunile `BT`/`BTS`/`BTR`/`BTC` nu afectează alte *flag*-uri, în afară de *flag*-ul *carry* (`CF`).

4.9. Instrucțiunile SETcc

Instrucțiunile `SETcc` (Set on Condition) setează un operand pe 8 biți (un octet), care poate fi registru de uz general sau locație de memorie, la valoarea zero (`0`) sau unu (`1`) în funcție de valorile *flag*-urilor. Formele generale pentru instrucțiunile `SETcc` sunt:

```
set_cc   reg_8
set_cc   mem_8
```

Sintaxa instrucțiunilor `SETcc` este compusă din cuvântul din engleză „SET" căruia i se anexează acronimul din engleză al codului de condiție (`cc` – din endl. Conditional Code) Instrucțiunile enumerate în Tabelul 4.6 ca perechi (de exemplu, `SETA/SETNBE`) sunt nume alternative ale aceleiași instrucțiuni pentru care asamblorul oferă nume alternative pentru a crește lizibilitatea codului.

`SETcc` reprezintă astfel o serie de mnemonici cu diferite semnificații care pot fi observate în Tabelul Tabelul 4.6. Aceste instrucțiuni stochează o valoare de `0` în operandul destinație, dacă condiția este falsă, sau o valoare de `1` dacă condiția este adevărată.

Tabelul 4.6. Instrucțiunile `SETcc`.

Mnemonica		*Descriere*	*Starea FLAG-urilor*
Evaluări fără semn	SETA/SETNBE	Above / Not Below or Equal	CF = 0 *and* ZF = 0
	SETAE/SETNB	Above or Equal / Not Below	CF = 0
	SETNC	Not Carry	
	SETC	Carry	CF = 1
	SETB/SETNAE	Below / Not Above or Equal	
	SETBE/SETNA	Below or Equal / Not Above	CF = 1 *or* ZF = 1
	SETP/SETPE	Parity / Parity Even	PF = 1
	SETNP/SETPO	Not Parity / Parity Odd	PF = 0
	SETE/SETZ	Equal / Zero	ZF = 1
	SETNE/SETNZ	Not Equal / Not Zero	ZF = 0
Evaluări cu semn	SETG/SETNLE	Greater / Not Less or Equal	SF = OF *or* ZF = 0
	SETGE/SETNL	Greater or Equal / Not Less	SF = OF
	SETL/SETNGE	Less / Not Greater or Equal	SF ≠ OF
	SETLE/SETNG	Less or Equal / Not Greater	SF ≠ OF *or* ZF = 1
	SETO	Overflow	OF = 1
	SETNO	Not Overflow	OF = 0
	SETS	Sign (negative)	SF = 1
	SETNS	Not Sign (non-negative)	SF = 0

Instrucțiunile SET*cc* pur și simplu testează *flag*-urile fără alte semnificații atașate operațiunii. De exemplu, instrucțiunea SETC pentru a verifica valoarea *flag*-ului *carry* modificat în urma executării unei instrucțiuni de deplasare, de testare și schimbare a biților sau operație aritmetică. De asemenea, instrucțiunea SETNZ poate fi utilizată după o instrucțiune TEST pentru a verifica rezultatul.

Instrucțiunea CMP poate fi folosită cu instrucțiunile setcc. Imediat după o operație CMP, *flag*-urile procesorului oferă informații cu privire la valorile relative dintre operanzii instrucțiunii CMP. Aceste flag-uri permit evaluarea condițiilor de egalitate sau inegalitate sau o combinație a acestora.

Există două grupuri de instrucțiuni SET*cc*. Primul grup reprezintă instrucțiunile care evaluează rezultatul pentru numere fără semn, iar al doilea grup pentru evaluări cu semn.

Instrucțiunile SET*cc* sunt deosebit de utile deoarece pot converti rezultatul unei comparații la o valoare booleană (adevărat/fals sau 0/1). Acest lucru este deosebit de important când se traduc expresii logice dintr-un limbaj de nivel înalt, cum C/C++, în limbaj de asamblare. De exemplu, dacă vrem să evaluăm o expresie de forma „X >= Y”, unde X și Y reprezintă două numere întregi cu semn pe 32 biți, secvența de instrucțiuni de mai jos realizează acest lucru, cu observația că variabila Bool trebuie să fie pe 8 biți (BYTE):

```
        mov     eax, X
        cmp     eax, Y
```

```
        setge  Bool
```

Deoarece instrucțiunile SETcc produc întotdeauna valori de zero sau unu, acestea pot fi utilitate împreună cu instrucțiunile logice AND și OR pentru a calcula valori booleene complexe:

```
        ; Bool = ((X >= Y) && (Z != T) || (Z == X))
        mov    eax, X
        cmp    eax, Y
        setge  cl
        mov    eax, Z
        cmp    eax, T
        setne  ch
        and    cl, ch
        mov    eax, Z
        cmp    eax, X
        sete   ch
        or     cl, ch
        mov    Bool, bh
```

4.10. Instrucțiuni cu șiruri

Instrucțiunile uzuale pentru operații cu șiruri sunt:

- MOVS (Move String)
- LODS (Load String)
- STOS (Store String)
- SCAS (Scan String)
- CMPS (Compare String Operands).

Aceste instrucțiuni acceptă prefixe de repetare a operațiilor cu șiruri:

- REP (Repeat)
- REPZ (Repeat While Zero)
- REPE (Repeat While Equal)
- REPNZ (Repeat While Not Zero)
- REPNE (Repeat While Not Equal).

Instrucțiunile MOVS, STOS, SCAS și CMPS sunt utilizate pentru a opera cu un singur element dintr-un șir sau pentru a procesa un întreg șir. În general, instrucțiunea LODS este utilizată pentru încărcarea unui singur element într-un șir.

Aceste instrucțiuni pot opera cu șiruri de octeți (BYTE), cuvinte (WORD), dublu-cuvânt (DWORD) sau cvadruplu-cuvânt (QWORD). Pentru a specifica dimensiunea obiectului, pur și simplu se adaugă litera B, W, D sau Q la sfârșitul mnemonicii instrucțiunii, de exemplu: LODSB, MOVSW, SCASD, SCASQ, etc..

Instrucțiunile MOV și SCAS utilizează conținutul registrul RSI ca adresă a șirului sursă și RDI ca adresă a șirului destinație. Instrucțiunea LODS utilizează ca adresă a șirului sursă, registrul RSI, iar registrul acumulator (AL/AX/EAX/RAX) este destinația implicită. Instrucțiunile SCAS și SCAS utilizează RDI ca adresă a șirului destinație și acumulatorul ca valoare sursă.

4.10.1. Instrucțiunea MOVS

Instrucțiunea MOVS (Move String) transferă un element al unui șir (BYTE, WORD, DWORD sau QWORD) din locația de memorie din RSI în locația de memorie din RDI. După copierea datelor, instrucțiunea actualizează adresele din RSI și RDI în funcție de valoarea *flag*-ului DF (Direction Flah) și a dimensiunii operanzilor. Astfel dacă *flag*-ul *direction* (DF) este 0 valorile regiștrilor RSI

și RDI vor fi incrementate cu 1, 2, 4 sau 8, altfel vor fi decrementate cu aceleași valori care reprezintă dimensiunea cantității de date transferate. Instrucțiunile STD și CLD pot fi utilizate pentru setarea respectiv ștergerea valoarea flag-ului DF.

Instrucțiunea MOVS are următoarele forme:

```
movs   mem_x, mem_x                                    ; X = 8/16/32/64
movsb
movsw
movsd
movsq
```

La nivel de cod de asamblare, sunt permise două forme ale acestei instrucțiuni: forma cu „operanzi expliciți" și forma „fără operanzi". Forma cu operanzi expliciți (specificată cu mnemonica MOVS) permite specificarea în mod explicit a celor doi operanzi (sursă și destinație). Aici, operanzii *ar trebui să fie* simboluri care indică mărimea și locația valorilor sursei. Această formă cu operanzi expliciți este furnizată pentru a permite documentarea codului. Cu toate acestea, trebuie reținut faptul că documentația furnizată de acestă formă poate fi înșelătoare. Adică, operanzii sursă specifică *doar tipul* (mărimea) corectă a operanzilor (octeți, cuvinte dublu-cuvinte sau cvadruplu-cuvinte), dar nu și locația efectivă. Cu alte cuvinte valoarea (adresa) operanzilor de memorie este ignorată. Locațiile operanzilor sursă și destinație sunt *întotdeauna* specificate de regiștrii RSI și RDI, care trebuie încărcate corect înainte de a executa instrucțiunea de transfer a datelor între șiruri.

Forma fără operanzi oferă „forme scurte" ale versiunilor octeți, cuvinte, dublu-cuvinte sau cvadruplu-cuvinte ale instrucțiunilor MOVS. Și pentru această formă regiștrii RSI și RDI sunt, operanzii sursă și destinație impliciți. Mărimea operanzilor sursă și destinație este selectată cu mnemonic: MOVSB (Byte Move String), MOVSW (Word Move String), MOVSD (Dword Move String) sau MOVSQ (Qword Move String).

Instrucțiunea MOVS mută blocuri de date în memorie. MOVS poate fi utilizată pentru a copia șiruri, tablouri și alte structuri de date cu mai mulți octeți (1, 2, 4 sau 8 octeți). Acțiunea instrucțiunii MOVS poate fi descrisă de secvența:

```
; MOVS{B/W/D/Q}
BYTE/WORD/DWORD/QWORD ptr [RDI] = [RSI]
if (DF == 0)
       {
         RSI = RSI + sizeof(BYTE/WORD/DWORD/QWORD)  ; 1/2/4/8
         RDI = RDI + sizeof(BYTE/WORD/DWORD/QWORD)  ; 1/2/4/8
       }
else   {
         RSI = RSI - sizeof(BYTE/WORD/DWORD/QWORD)
         RDI = RDI - sizeof(BYTE/WORD/DWORD/QWORD)
       }
```

Instrucțiunile MOVS, MOVSB, MOVSW, MOVSD și MOVSQ pot fi precedate de prefixul REP (a se vedea secțiunea 4.10.6, de la pagina 105) pentru a transfera blocuri de RCX octeți, cuvinte, dublu-cuvinte sau cvadruplu-cuvinte.

4.10.2. Instrucțiunea CMPS

Instrucțiunea CMPS (Compare String Operands) compară octetul, cuvântul, dublu-cuvântul sau cvadruplu-cuvânt de la adresa indicată de registrul RSI cu valoarea de la adresa din RDI și stabilește flag-urile procesorului în consecință. După realizarea comparației, CMPS crește sau

descrește RSI și RDI cu 1, 2, 4 sau 8, în funcție de dimensiunea instrucțiunii și de starea *flag*-ului DF din registrul EFLAGS.

Formele instrucțiunii sunt:

```
cmps   mem_x, mem_x                                    ; X = 8/16/32/64
cmpsb
cmpsw
cmpsd
cmpsq
```

Ca și instrucțiunea MOVS, CMPS are două forme: forma cu „operanzi expliciți” și forma „fără operanzi”. În forma cu operanzi impliciți (forma CMPS), operanzii locații de memorie pe 8, 16, 32 sau 64 biți sunt utilizați doar pentru a specifica dimensiunea operației. Locațiile operanzilor sursă sunt întotdeauna specificate de regiștrii RSI și RDI, care trebuie încărcate corect înainte de a executa instrucțiunea de comparație a șirurilor.

Forma fără operanzi oferă „forme scurte” ale versiunilor pe octeți, cuvinte, dublu-cuvinte sau cvadruplu-cuvinte ale instrucțiunilor CMPS. Aici, de asemenea, procesorul utilizează regiștrii RSI și RDI pentru a specifica locația operanzilor sursă. Dimensiunea operanzilor sursă este selectată prin mnemonică: CMPSB (comparație de octeți), CMPSW (comparație de cuvinte), CMPSD (comparație de dublu-cuvinte) sau CMPSQ (comparație de cvadruplu-cuvinte).

Operația instrucțiunii CMPS poate fi descrisă de secvența:

```
; CMPS{B/W/D/Q}
cmp BYTE/WORD/DWORD/QWORD ptr [RDI], [RSI]
if (DF == 0)
       {
         RSI = RSI + sizeof(BYTE/WORD/DWORD/QWORD)  ; 1/2/4/8
         RDI = RDI + sizeof(BYTE/WORD/DWORD/QWORD)  ; 1/2/4/8
       }
else   {
         RSI = RSI - sizeof(BYTE/WORD/DWORD/QWORD)
         RDI = RDI - sizeof(BYTE/WORD/DWORD/QWORD)
       }
```

Instrucțiunile CMPS, CMPSB, CMPSW, CMPSD și CMPSQ pot fi precedate de prefixul REPE/REPZ sau REPNE/REPNZ pentru a realiza comparații de blocuri (mai multe detalii în secțiunea 4.10.6, de la pagina 105). Totuși, mai des, aceste instrucțiuni vor fi utilizate într-o construcție de tip LOOP pentru a lua anumite decizii bazate pe setarea *flag*-urilor înainte de a face următoarea comparație.

4.10.3. Instrucțiunea LODS

Instrucțiunea LODS (Load String) copiază octetul, cuvântul, dublu-cuvântul sau cvadruplu-cuvântul de la adresa indicată de registrul RSI în acumulator: AL, AX, EAX respectiv RAX. Ulterior valoarea din RSI va fi incrementată sau decrementată cu 1, 2, 4 sau 8, în funcție de dimensiunea instrucțiunii și valoarea *flag*-ului DF. Instrucțiunea LODS este utilă pentru a obține o secvență de octeți, cuvinte dublu-cuvinte sau cvadruplu-cuvinte dintr-un tablou, efectuând unele operații pe acele valori și mai apoi se va trece la următorului element din șir.

Formele instrucțiunii LODS sunt:

```
lods   mem_x                                           ; X = 8/16/32/64
lodsb
lodsw
lodsd
```

```
lodsq
```

Similar instrucțiunilor descrise anterior, instrucțiunea LODS are două forme: forma „operand explicit” și cea „fără operand”. Forma cu operand (specificată cu mnemonica LODS) permite specificarea operandului sursă în mod explicit, care are doar rolul de a indica dimensiunea valorii sursei. Operandul de destinație este apoi selectat automat pentru a se potrivi cu dimensiunea operandului sursă (registrul AL pentru operanzi de octeți, AX pentru operanzi de cuvânte, EAX pentru operanzi de dublu-cuvinte și RAX pentru operanzi de cvadruplu-cuvinte). Locația sursei este specificată întotdeauna de registrul RSI, care trebuie încărcate corect înainte de executarea instrucțiunii LODS.

Forma fără operand oferă „forme scurte” ale versiunilor pe octeți, cuvinte, dublu-cuvinte sau cvadruplu-cuvinte ale instrucțiunilor LODS. Aici, de asemenea, RSI este operandul sursă implicit, iar registrul AL, AX, EAX sau RAX este operandul destinație. Mărimea operanzilor sursă și destinație este selectată cu mnemonic: LODSB (octet încărcat în registrul AL), LODSW (cuvânt încărcat în AX), LODSD (dublu-cuvântul încărcat în EAX) sau LODSQ (cvadruplu-cuvânt încărcat în EAX).

Operația instrucțiunii LODS poate fi descrisă de secvența:

```
; LODS{B/W/D/Q}
AL/AX/EAX/RAX = [RSI]
if (DF == 0)
       RSI = RSI + sizeof(BYTE/WORD/DWORD/QWORD)  ; 1/2/4/8
else
       RSI = RSI - sizeof(BYTE/WORD/DWORD/QWORD)
```

Instrucțiunile LODS, LODSB, LODSW și LODSD pot fi precedate de prefixul REP pentru a încărca blocuri de RCX octeți, cuvinte dublu-cuvinte sau cvadruplu-cuvinte (a se vedea secțiunea 4.10.6, de la pagina 105). Totuși, mai des, aceste instrucțiuni sunt utilizate în cadrul unei construcții LOOP, deoarece prelucrarea ulterioară a datelor mutate în registrul acumulator este de obicei necesară înainte de a efectua următorul transfer.

4.10.4. Instrucțiunea STOS

Instrucțiunea STOS (Store String) stochează valoarea din registrul AL, AX, EAX sau RAX la adresa specificată de registrul RDI. Ulterior, RDI este incrementat sau decrementat în funcție de dimensiunea instrucțiunii și valoarea flag-ului DF. Instrucțiunea STOS are mai multe utilizări. Împreună cu instrucțiunile LODS (tratate anterior), se poate încărca (cu LODS), manipula și stoca elemente șirurilor.

Instrucțiunea STOS poate avea una din formele:

```
stos   mem_x                                          ; X = 8/16/32/64
stosb
stosw
stosd
stosq
```

Instrucțiunea STOS are, de asemenea, două forme. Forma cu operand (specificată cu mnemonica STOS) permite specificarea explicită a operandului de destinație. Acest operand este utilizat doar pentru a specifica tipul (dimensiunea) operandului (octet, cuvânt, dublu-cuvânt sau cvadruplu-cuvânt), dar nu va conține locația corectă. Locația este specificată întotdeauna de registrul RDI. Acestea trebuie încărcat înainte de executarea instrucțiunilor STOS.

Forma fără operand furnizează „forme scurte” ale versiunilor de octeți, cuvinte, dublu-

cuvinte sau cvadruplu-cuvinte ale instrucțiunilor STOS. Dimensiunea operandului destinație și sursa este selectată de mnemonic: STOSB (octet citit din registrul AL), STOSW (cuvântul din AX), STOSD (dublu-cuvântul din EAX) sau STOSQ (cvadruplu-cuvânt din RAX). Registrul RDI este operandul destinație implicit iar AL, AX, EAX sau RAX este operandul sursă.

Comportamentul acestei instrucțiuni este descris de secvența:

```
; STOS{B/W/D/Q}
[RDI] = AL/AX/EAX/RAX
if (DF == 0)
        RDI = RDI + sizeof(BYTE/WORD/DWORD/QWORD)  ; 1/2/4/8
else
        RDI = RDI - sizeof(BYTE/WORD/DWORD/QWORD)
```

Utilizată fără prefix instrucțiunea STOS poate stoca rapid o singură valoare într-o structură de date. Instrucțiunile STOS pot fi precedate de prefixul REP pentru a încărcă un întreg bloc de RCX octeți, cuvinte, dublu-cuvinte sau cvadruplu-cuvinte (a se vedea secțiunea 4.10.6, de la pagina 105). Totuși, adesea, aceste instrucțiuni sunt utilizate în cadrul unei construcții LOOP, deoarece datele trebuie mutate în registrul AL, AX, EAX sau RAX înainte de a putea fi stocate. De asemenea, trebuie menționat că o instrucțiune compusă „REP STOS” este cea mai rapidă modalitate de inițializare a unui bloc de memorie.

4.10.5. Instrucțiunea SCAS

Instrucțiunea SCAS compară valoarea din registrul acumulator AL, AX, EAX sau RAX cu valoarea de la adresa din RDI după care ajustează adresa din RDI în consecință. Această instrucțiune stabilește flag-urile procesorului la fel ca instrucțiunile CMP și SCAS. Instrucțiunea SCAS este excelentă pentru căutarea unei anumite valori într-o structură de date.

Instrucțiunea SCAS are formele:

```
scas   mem_x                                        ; X = 8/16/32/64
scasb
scasw
scasd
scasq
```

Ca și celelalte instrucțiuni cu șiruri, instrucțiunea SCAS are cele două forme: forma cu operand (mnemonica SCAS) care este utilizat doar pentru a specifica tipul (dimensiunea) operandului și formele fără operand cu mnemonicele: SCASB (comparație de octeți), SCASW (comparație de cuvinte), SCASD (comparație de dublu-cuvinte) sau SCASQ (comparație de cvadruplu-cuvinte). Registrul RDI este operandul memorie implicit iar AL, AX, EAX sau RAX este registrul operand.

Operația instrucțiunii SCAS este descrisă mai jos:

```
; SCAS{B/W/D/Q}
cmp AL/AX/EAX/RAX, [RDI]
if (DF == 0)
        RDI = RDI + sizeof(BYTE/WORD/DWORD/QWORD)  ; 1/2/4/8
else
        RDI = RDI - sizeof(BYTE/WORD/DWORD/QWORD)
```

Instrucțiunile SCAS, SCASB, SCASW, SCASD și SCASQ pot fi precedate de prefixul REPE/REPZ sau REPNE/REPNZ pentru compararea blocurilor de RCX octeți, cuvinte, dublu-cuvinte sau cvadruplu-cuvinte (a se vedea secțiunea 4.10.6, de mai jos). Adesea, aceste instrucțiuni vor fi

utilizate într-o structură LOOP în care se vor efectua operații pe baza valorilor *flag*-urilor setate în urma unei comparații.

4.10.6. Prefixe de repetare a operațiilor cu șiruri

Fiecare dintre instrucțiunile cu șiruri, descrise anterior, efectuează o iterație a unei operații de șir. Pentru a opera pe șiruri mai lungi decât un cvadruplu-cuvânt, instrucțiunile cu șiruri pot fi combinate cu un prefix de repetare (REP) pentru a o instrucțiune care se repetă sau pot fi plasate într-o buclă repetitivă.

Când sunt utilizate de către instrucțiunile cu șiruri, regiștrii RSI și RDI sunt incrementate sau decrementate automat după fiecare iterație a unei instrucțiuni pentru a indica următorul element (octet, cuvânt, dublu-cuvânt sau cvadruplu-cuvânt) din șir. Operațiile cu șiruri pot începe astfel de la adrese superioare și pot opera către cele inferioare sau invers. *Flag*-ul DF din registrul EFLAGS controlează dacă regiștrii sunt incrementați (DF=0) sau decrementați (DF=1). Instrucțiunile STD și CLD pot fi utilizate pentru a modifica direct acest *flag*.

Dacă o instrucțiune cu șiruri este precedată de REP, REPE/REPZ sau REPNE/REPNZ (Repeat String Operation Prefix) aceasta va fi repetată de numărul de ori specificat în registrul RCX sau cât timp este îndeplinită condiția indicată a *flag*-ului ZF. Mnemonicele REP (Repeat), REPE (Repeat while Equal), REPNE (Repeat while Not Equal), REPZ (Repeat while Zero), și REPNZ (Repeat while Not Zero) sunt prefixe care pot fi adăugate la una dintre instrucțiunile cu șirului. Prefixul REP poate fi adăugat instrucțiunilor MOVS, LODS și STOS, iar prefixurile REPE/REPZ, REPNE/REPNZ și pot fi adăugate la instrucțiunile CMPS și SCAS. (Prefixele REPZ și REPNZ sunt forme sinonime ale prefixelor REPE, respectiv REPNE).

Prefixele de repetare se aplică numai unei singure instrucțiuni cu șiruri. Pentru a repeta un bloc de instrucțiuni, poate fi utilizată instrucțiunea LOOP sau oricare alt tip de buclă repetitivă. Toate aceste prefixe de repetare fac ca instrucțiunea asociată (precedată) să fie repetată până când contorul din registrul RCX este decrementat la 0. Când o instrucțiune cu șiruri are un prefix de repetare, operația se execută până când una dintre condițiile de terminare specificate de prefix este satisfăcută (a se vedea tabelul 4.7).

Tabelul 4.7. Condițiile de terminare pentru prefixele de repetare a operațiilor cu șiruri.

Prefixul	*Condiția de terminare 1*	*Condiția de terminare 2*
REP	RCX == 0	-
REPE/REPZ		ZF == 0
REPNE/REPNZ		ZF == 1

4.11. Instrucțiuni de control al fluxului programului

Instrucțiunile discutate în sub-capitolele anterioare se execută secvențial; adică procesorul execută fiecare instrucțiune din secvența în ordinea în care acestea apar în program. Scrierea de programe reale necesită mai multe structuri de control, nu execuția secvențială. Exemple includ decizia, buclele repetitive și apelurile de subrutine. Deoarece compilatoarele reduc toate celelalte limbaje de programare la limbajul de asamblare, nu ar trebui să surprindă faptul că limbajul de asamblare acceptă instrucțiunile necesare pentru implementarea acestor structuri de control. Instrucțiunile de control al programelor pentru arhitectura x86 se împart în trei grupuri: salturi necondiționate, salturi condiționale și instrucțiuni de apelare și revenire din subrutine. Următoarele secțiuni descriu aceste instrucțiuni.

4.11.1. Instrucțiunea JMP

Instrucțiunea `JMP` (Jump) transferă necondiționat controlul programului la o instrucțiune de destinație. Transferul este într-un singur sens – adică o adresă de revenire nu este salvată. Un operand destinație specifică adresa instrucțiunii destinație. Adresa poate fi o adresă relativă sau o adresă absolută.

O adresă relativă reprezintă un deplasament (offset) față de adresa din registrul `RIP` (care conține adresa instrucțiunii imediat următoare). Adresa destinație (pointer de tip *near*) este formată prin adăugarea deplasamentului la adresa din registrul `RIP`. Deplasamentul este specificat ca un număr întreg cu semn, permițând salturi înainte sau înapoi în secvența de instrucțiuni.

O adresă absolută, pentru sistemele pe 64 biți, reprezintă o valoare care poate fi specificată în următoarele moduri:

- O adresă dintr-un registru de uz general. Această adresă este tratată ca un pointer de tip *near*, care este copiată în registrul `RIP`. Execuția programului continuă apoi la noua adresă.
- O adresă specificată folosind modurile de adresare standard ale procesorului. Aici, adresa poate fi de tip *near* sau de tip *far*. Dacă adresa este o locație de memorie pe 64 biți, aceasta este un pointer de tip *near* și va fi copiată în registrul `RIP`. Dacă adresa este pentru un pointer *far*, adresa este tradusă într-un selector de segment (primii 16 biți) care este copiat în registrul `CS` și un deplasament (care este copiat în registrul `RIP`).

Formele instrucțiunii `JMP` sunt:

```
jmp     rel8            ; short relative direct jump (8 bit offset)
jmp     rel32           ; near relative direct jump (32 bit offset)
jmp     reg64           ; register alsolute indirect near jump
jmp     mem64           ; memory absolute indirect near jump
jmp     mem16:mem16     ; memory absolute indirect far jumps
jmp     mem16:mem32
jmp     mem16:mem64
```

Această instrucțiune poate fi folosită pentru a executa patru tipuri diferite de salturi:

- *near jump* - salt la o instrucțiune din segmentul de cod curent (segmentul indicat de registrul `CS`), uneori denumit și salt intra-segment;
- *short jump* - un salt apropiat în care intervalul de salt este limitat la gama de la –128 până la +127 din valoarea curentă a registrului `RIP`;
- *far jump* - salt la o instrucțiune situată într-un segment diferit de segmentul de cod curent, dar la același nivel de privilegiu, uneori denumit salt de inter-segment;
- *task switch* - un salt la o instrucțiune situată într-un fir de execuție diferit.

Primele două forme folosesc o schemă de adresare relativă. Deplasamentul (offset) nu este adresa țintă din segmentul codului curent, ci distanța relativă până la adresa țintă. Asamblorul va calcula automat distanța, astfel încât nu trebuie calculă de către programator această valoare. În multe privințe, aceste instrucțiuni nu sunt altceva decât: „`add rip, rel`$_{8/32}$”.

Prima formă are ca operand un deplasament de un singur octet. Procesorul va extinde cu semn această valoare la 64 biți și o va adaugă la registrul `RIP`. Această instrucțiune va executa un salt la o locație cu -128 până la +127 față de începutul instrucțiunii imediat următoare (adică de la -126 până la +129 octeți față de instrucțiunea de salt curentă care este codată pe 2 octeți).

A doua formă a saltului intra-segment are o lungime de cinci octeți și conține un deplasament pe 32 biți. Această instrucțiune permite un salt în gama -32768 ... +32767 de octeți

față de instrucțiune imediat următoare. Procesorul pur și simplu adaugă deplasamentul extins cu semn la 64 biți la valoare curentă a registrului RIP (care la momentul execuției instrucțiunii de salt va conține adresa instrucțiunii imediat următoare acesteia).

Pentru cele două salturi directe descrise mai sus, în mod normal, programatorul specifică adresa țintă folosind o etichetă a instrucțiunii țintă. O etichetă este de obicei un identificator urmat de două puncte, de obicei pe aceeași linie cu o instrucțiune. Asamblorul determină deplasamentul funcție de această etichetă și calculează automat distanța de la instrucțiunea de salt la eticheta instrucțiunii.

Cea de-a treia și a patra formă sunt forme de salt indirect intra-segment și vor copia în registrul RIP valoarea din registrul de uz general sau din locația de memorie pe 64 biți specificată.

Ultimele trei forme ale instrucțiunii JMP transferă necondiționat controlul la o altă adresă, fără a salva valorile curente ale perechii de regiștri CS:RIP. Această formă a instrucțiunii sare la o adresă din afara segmentului de cod curent și este denumită *far jump*. Operandul specifică un selector și o deplasament către o adresă țintă. În modul pe 64 biți operandul țintă este specificat indirect, acesta reprezentând adresa unei locații pe 16 biți care va conține selectorul care va fi încărcat în registrul CS, urmat un deplasament pe 16, 32 sau 64 biți care va fi extins cu semn la 64 biți și încărcat în registrul RIP.

Instrucțiunea JMP nu modifică *flag*-urile

4.11.2. Instrucțiunile CALL și RET

Instrucțiunile CALL și RET (Return) gestionează apelurile către subrutine, respectiv revenirile din subrutine. Formele instrucțiunii CALL sunt:

```
call   rel32          ; near relative direct call (32 bit offset)
call   reg64          ; register alsolute indirect near call
call   mem64          ; memory absolute indirect near call
call   mem16:mem16    ; memory absolute indirect far calls
call   mem16:mem32
call   mem16:mem64
```

Instrucțiunea CALL adoptă aceleași forme ca și instrucțiunea JMP, cu excepția faptului că nu există un apel cu offset pe 8 biți (un octet). Astfel instrucțiunea CALL oferă doar 3 tipuri de apeluri *near* și *far* (JMP avea și *short jump*). Spre deosebite de intrucțiunea JMP care efectuează un salt într-un singur sens, perechile de instrucțiuni CALL și RET realizează apelul și revenirea din proceduri printr-un mecanism de salvare respectiv restaurare pe/de pe stivă a unei adrese de revenire.

Apelurile *near* urmează următorii pași:

- salvează pe stivă adresa (pe 64 biți) a instrucțiunii imediat următoare (adresa de revenire);
- încarcă adresa specificată de operand în registrul RIP (instrucțiunea CALL permite aceleași moduri de adresare ca și JMP, apelul poate obține adresa țintă folosind un mod de adresare relativ direct sau indirect absolut prin memorie sau regiștri de uz general);
- execuția continuă la prima instrucțiune a subrutinei (această primă instrucțiune este cea specificată de adresa țintă calculată în pasul anterior).

Apelurile *far* salvează pe stivă mai întâi valoarea din registrul CS, după care urmează aceeași pași (prezentați anterior) ca și apelurile *near*.

Instrucțiunea RET are următoarele forme:

```
ret
retf
ret    imm16
retf   imm16
```

Instrucțiunea `RET` returnează controlul codului apelant al unei subrutine. Acest lucru se realizează prin descărcarea adresei de revenire din vârful stivei și transferul controlului la instrucțiunea de la această adresă de revenire. O instrucțiune de revenire intra-segment (*near*). `RET`, încarcă în registrul `RIP` adresa (pe 64 biți) din vârful stivei. O revenire inter-segment (`far`), `RETF`, încarcă în registrul `RIP` adresa (pe 64 biți) din vârful stivei și și suplimentat față de varianta *near* va încărca și un selector de 16 biți de pe stivă în registrul `CS`. În procesul de extragere a selectorului de segment de pe stivă se vor citi 64 biți din vârful stivei dintre care cei mai nesemnificativi 16 vor fi încărcați în registrul `CS`, restul de 48 biți mai semnificativi vor fi ignorați.

Este de la sine înțeles că revenirea dintr-o rutină de tip *near* trebuie să se facă cu un o revenire de tip *near* (`RET`), iar un apel de subrutină *far* cu un `RETF` corespunzător. Dacă se amestecă apelurile *near* cu reveniri *far* sau invers, cel mai probabil nu se va reveni la instrucțiunea corespunzătoare după apel. Desigur, o altă problemă importantă atunci când sunt utilizate instrucțiunile `CALL` și `RET` este aceea că programatorul trebuie să se asigure că înainte de apelul unei instrucțiuni `RET` în vârful stivei trebuie să se găsească adresa de revenire salvată de apelul `CALL`. Pentru acesta o subrutina trebuie să elibereze stiva alocată pe parcursul execuției acesteia înainte de a încerca revenirea la codul apelant. Managementul defectuos al stivei constituie o cauză majoră a erorilor în subrutinele scrise în limbaj de asamblare.

Ultimele două forme ale instrucțiunii `RET` sunt identice cu primele două, cu excepția operandului imediat pe 16 biți care urmează codurile lor. Procesorul adaugă această valoare la registrul `RSP` imediat după extragerea adresei de revenire de pe stivă.

Instrucțiunile `CALL` și `RET` nu modifică indicatorii de condiție din `RFLAGS`.

4.11.3. Instrucțiunile Jcc

Deși instrucțiunile de `JMP`, `CALL` și `RET` oferă mecanisme de transfer al controlului programului, acestea nu permit să luarea de decizii. Instrucțiunile de salt condiționat ale procesorului x86–64 gestionează această sarcină. Instrucțiunile de salt condiționat sunt instrumentul de bază pentru crearea de bucle repetitive și alte structuri de decizie.

Salturile condiționale testează unul sau mai multe *flag*-uri din registrul `FLAGS` pentru a vedea dacă se potrivesc cu un anumit tipar (la fel ca instrucțiunile `SETcc`). Dacă tiparul se potrivește, controlul este transferat locației țintă. Dacă tiparul nu reușește, procesorul ignoră saltul condiționat și execuția continuă cu instrucțiunea imediat următoare. Unele instrucțiuni, de exemplu, testează condițiile de semn (`SF`), transport (`CF`), depășire (`OF`) și a indicatorului de zero (`ZF`). De exemplu, după executarea unei instrucțiuni de `SHL`, se poate testa *flag*-ul *carry* pentru a determina dacă bitul cel mai semnificativ care se pierde în urma operației de *shift*-are este `1`. De asemenea, starea *flag*-ului *zero* poate fi testat după după o instrucțiune `TEST` pentru a vedea dacă între biții specificați măcar unul are valoarea `1`. Cu toate acestea, de cele mai multe ori, probabil, instrucțiunile de salt condiționat vor urma o instrucțiune `CMP`.

Sintaxa instrucțiunilor `Jcc` este compusă din litera „`J`” (de la cuvântul din engleză *Jump*) căreia i se anexează acronimul din engleză al codului de condiție. Documentația Intel definește diverse sinonime sau mnemonici echivalente pentru multe dintre instrucțiunile de salt condiționat. Următoarele tabele prezintă aceste echivalențe pentru o anumită instrucțiune dar și instrucțiunea opusă.

Tabelul 4.8. Instrucțiuni `Jcc` pentru testarea *flag*-urilor.

Mnemonica	Descriere	Condiția	Sinonim	Antonim
`JC`	Jump if carry	`CF=1`	`JB, JNAE`	`JNC`
`JNC`	Jump if no carry	`CF=0`	`JNB, JAE`	`JC`
`JZ`	Jump if zero	`ZF=1`	`JE`	`JNZ`
`JNZ`	Jump if not zero	`ZF=0`	`JNE`	`JZ`
`JS`	Jump if sign	`SF=1`		`JNS`
`JNS`	Jump if no sign	`SF=0`		`JS`
`JO`	Jump if overflow	`OF=1`		`JNO`
`JNO`	Jump if no overflow	`OF=1`		`JO`
`JP`	Jump if parity	`PF=1`	`JPE`	`JNP`
`JPE`	Jump if even		`JP`	`JPO`
`JNP`	Jump if no parity	`PF=0`	`JPO`	`JP`
`JPO`	Jump if odd		`JNP`	`JPE`

Tabelul 4.9. Instrucțiunile `Jcc` pentru comparații fără semn.

Mnemonica	Descriere	Condiția	Sinonim	Antonim
`JA`	Jump if above (>)	`CF=0` și `ZF=0`	`JNBE`	`JNA`
`JNBE`	Jump if not below or equal (not ≤)		`JA`	`JBE`
`JAE`	Jump if above or equal (≥)	`CF=0`	`JNC, JNB`	`JNAE`
`JNB`	Jump if not below (not <)		`JNC, JAE`	`JB`
`JB`	Jump if below (<)	`CF=1`	`JC, JNAE`	`JNB`
`JNAE`	Jump if not above or equal (not ≥)		`JC, JB`	`JAE`
`JBE`	Jump if below or equal (≤)	`CF=1` sau `ZF=1`	`JNA`	`JNBE`
`JNA`	Jump if not above (not >)		`JBE`	`JA`
`JE`	Jump if equal (=)	`ZF=1`	`JZ`	`JNE`
`JNE`	Jump if not equal (≠)	`ZF=0`	`JNZ`	`JE`

Tabelul 4.10. Instrucțiunile `Jcc` pentru comparații cu semn.

Mnemonica	Descriere	Condiția	Sinonim	Antonim
`JG`	Jump if greater (>)	`SF=OF` sau `ZF=0`	`JNLE`	`JNG`
`JNLE`	Jump if not less or equal (not ≤)		`JG`	`JLE`
`JGE`	Jump if greater or equal (≥)	`SF=OF`	`JNL`	`JNGE`
`JNL`	Jump if not less (not <)		`JGE`	`JL`
`JL`	Jump if less (<)	`SF≠OF`	`JNGE`	`JNL`

JNGE	Jump if not greater or equal (not ≥)		JL	JGE
JLE	Jump if less or equal (≤)	SF≠OF sau ZF=1	JNG	JNLE
JNG	Jump if not greater (not >)		JLE	JG
JE	Jump if equal (=)	ZF=1	JZ	JNE
JNE	Jump if not equal (≠)	ZF=0	JNZ	JE

Instrucțiunile Jcc au câte două forme:

```
jcc     rel8
jcc     rel32
```

Spre deosebire de instrucțiunea de salt necondiționat JMP instrucțiunile de salt condiționat, Jcc, au doar formele de salt relativ direct cu un desplasament de 8 (*short*) sau 32 biți (*near*). Instrucțiunea Jcc nu acceptă salturi de tip *far* (salturi către alte segmente de cod). Când ținta pentru saltul condițional se află într-un alt segment, se poate utiliza în schimb condiția inversă față de condiția testată pentru instrucțiunea Jcc, urmată de un JMP de tip far pentru a executa un salt necondiționat către un alt segment.

De exemplu instrucțiunea (ilegală) de salt condiționat de tip *far* de forma:

```
jz      FarLabel            ; ilegal, doar in scop demonstrativ
```

poate fi înlocuită de secvența de mai jos:

```
        jnz     @F
        jmp     FarLabel
@@:
```

După cum se poate observa în tabelele de mai sus, instrucțiunile de salt condiționat au sinonime (mnemonici echivalente). Aceasta înseamnă că există sinonime și pentru salturile „inverse”. Cu doar două excepții, se poate observa o regulă foarte simplă care descrie modalitatea de „obținere” a menmonicii opuse:

- dacă a doua literă a instrucțiunii Jcc nu este litera „n”, introduceți un „n” după „j” (de exemplu: jz devine jnz și ja devine jna, etc.);
- dacă a doua literă a instrucțiunii jcc este un „n”, atunci eliminați acel „n” din mnenonica instrucțiune (de exemplu: jnl devine jl, jno devine jo, etc.).

Cele două excepții de la această regulă sunt JPE (Jump if Parity Even) și JPO (Jump if Parity Odd). Aceste excepții provoacă puține probleme, deoarece pe de o parte flag-ul de paritate este foarte puțin utilizat în prezent, iar pe de altă parte se pot utiliza menmonicile echivalente JP și JNP sinonime pentru JPE și JPO, pentru care regula se mai sus se aplică.

Deși dacă, de exemplu, JGE este opusul lui JL, este recomandată utilizarea menonicii echivalente JNL și nu JGE. Deoarece în clasele primare la aritmetică elevii sunt obișnuiți cu sintagma „mai mare este opusul lui mai mic”, o greșeală des întâlnită în programarea în limbaj de asamblare este aceea că din reflex operația inversă a instrucțiunii este considerată mnenonica JG și nu varianta corectă JGE. Această confuzie poate fi evitată folosind întotdeauna regula prezentată mai sus.

La fel ca instrucțiunile SETcc, instrucțiunile de salt condiționat Jcc se încadrează în două categorii de bază – cele care testează valorile specificate ale flag-urilor procesului (de exemplu, JZ, JC, JNO) și cele care testează o anumită condiție (mai mic decât, mai mare decât, etc.). La testarea unei condiții, instrucțiunile Jcc succede aproape întotdeauna o instrucțiune CMP. Instrucțiunea CMP setează *flag*-urile, astfel încât o instrucțiune JA, JAE, JB, JBE, JE sau JNE poate

fi utilizată pentru a testa pentru condițiile de mai mic, mai mic sau egal, egalitate, negalitate, mai mare sau mai mare sau egal pentru numere fără semn. Similar, instrucțiunea `CMP` setează *flag*-urile, astfel încât să se poată efectua o comparație cu numere cu semn folosind instrucțiunile `JL`, `JLE`, `JE`, `JNE`, `JG` și `JGE`.

Instrucțiunile de salt condiționat testează *flag*-urile, și nu afectează nici unul din *flag*-urile procesorului x8-64.

4.11.4. Instrucțiunile JECXZ/JRCXZ

Instrucțiunile `JECXZ` (Jump if ECX is Zero) și `JRCXZ` (Jump if RCX is Zero) sunt tot instrucțiuni de salt condiționat, dar spre deosebire de instrucțiunile `Jcc`, acestea nu testează valorile *flag*-urilor. În schimb acestea verifică dacă valoarea registrului `ECX`, respectiv `RCX` este `0`. Aceste instrucțiuni sunt utile atunci când sunt utilizate la începutul unei bucle care se încheie cu o instrucțiune de buclă condițională (cum ar fi `LOOPNE`). Acestea pot fi utilizate pentru a împiedica o secvență de instrucțiuni să intre într-o buclă atunci când RCX sau ECX este 0. Aceasta ar determina bucla să execute de 2^{64} respectiv 2^{32} ori (și nu de zero ori).

Spre deosebire de celelalte instrucțiuni `Jcc` acestea au doar câte o formă:

```
jecxz  rel8
jrcxz  rel8
```

Dacă de exemplu considerăm instrucțiunea:

```
jecxz  ShortLabel
```

aceasta poate fi înlocuită de secvența:

```
test   ecx, ecx
jz     ShortLabel
```

Instrucțiunea `TEST` de mai sus va seta *flag*-ul *zero* dacă și numai dacă `ECX` are valoarea zero. De fapt, această secvență de două instrucțiuni este mai rapidă decât instrucțiunea `JECXZ` de pe procesoarele moderne. Într-adevăr, Intel recomandă utilizarea acestei secvențe, mai degrabă decât instrucțiunea `JECXZ` (și similar pentru `JRCXZ`) dacă se dorește optimizarea codului pentru viteză. Desigur, instrucțiunea `JECXZ/JRCXZ` este mai scurtă decât cele două secvențe de instrucțiuni, dar nu este mai rapidă. Acesta este un bun exemplu de excepție de la regula „mai scurt este de obicei mai rapid”.

Instrucțiunile `JECXZ/JRCXZ` nu afectează nici un *flag*.

4.11.5. Intrucțiunile LOOP/LOOPcc

Intrucțiune `LOOP` (Loop According to RCX Counter) este utilizată pentru a implementa bucle repetitive ce utilizează registrul `RCX` drept contor.

Intrucțiunile `LOOP/LOOPcc` au următoarele forme:

```
loop   rel8
loope  rel8            ; sau menmonica echivalenta LOOPZ
loopne rel8            ; sau menmonica echivalenta LOOPNZ
```

De fiecare dată când se execută instrucțiunea `LOOP`, registrul `RCX` este decrementat, apoi se verifică dacă este `0`. Dacă numărul acestuia este `0`, bucla este încheiată și execuția programului continuă cu instrucțiunea imediat următoare instrucțiunii `LOOP`. Dacă valoarea este diferită de zero, se va executa un *short jump* către operandul destinație (țintă), care este instrucțiunea de la începutul buclei.

Instrucțiunea țintă este specificată cu un deplasament relativ (o valoare pe 8 biți cu semn față de valoarea curentă a registrului RIP). Acest deplasament este, în general, specificat ca etichetă în codul de asamblare, dar la nivelul codului mașinii, este codată ca o valoare imediată cu semn, pe 8 biți, care este adăugată la registrului RIP. Deplasamente între -128 și +127 sunt permise cu această instrucțiune.

Unele forme ale instrucțiunii (LOOP*cc*) acceptă, de asemenea, *flag*-ul ZF ca o condiție pentru a încheia bucla înainte ca valoarea registrului RCX să ajungă la zero. Cu aceste forme ale instrucțiunii, un cod de condiție (*cc*) este asociat cu fiecare instrucțiune pentru a indica starea testată. Instrucțiunea LOOP*cc* în sine nu afectează starea flag-ului ZF, acesta este schimbat prin intermediul altor instrucțiuni din bucla LOOP*cc*.

Tabelul 4.11 conține descrierea funcționării instrucțiunilor LOOP/LOOP*cc*.

Tabelul 4.11: Instrucțiunile LOOP/LOOP*cc*.

Instrucțiunea	Descriere
LOOP	RCX--; *short jump* dacă RCX ≠ 0
LOOPE/LOOPZ	RCX--; *short jump* dacă RCX ≠ 0 și ZF = 1
LOOPNE/LOOPNZ	RCX--; *short jump* dacă RCX ≠ 0 și ZF = 0

Instrucțiunile LOOP/LOOP*cc* nu afectează *flag*-urile procesorului.

4.12. Instrucțiuni diverse

Există diverse instrucțiuni ale procesorului x86-64 care nu se încadrează în nici una din categorie de mai sus. În general, acestea sunt instrucțiuni care manipulează *flag*-urile, instrucțiuni pentru generarea de numere aleatorii, citirea contorului pentru timpul de procesare, non-instrucțiunea sau instrucțiunea de identificare a procesorului.

4.12.1. Instrucțiuni cu flagurile CF și DF

Există mai multe instrucțiuni care manipulează direct *flag*-urile CF și DF din registrul RFLAGS al procesorului x86-64.

Instrucțiunile STC (Set Carry Flag), CLC (Clear Carry Flag) și CMC (Complement Carry Flag) permit modificarea directă a *flag*-ului CF din registrul de stare RFLAGS. Aceste instrucțiuni nu au operanzi și permit setarea *flag*-ului carry pe 1 (STC), resetarea acestuia la 0 (CLC) și complementarea valorii acestuia (CLC). Acestea sunt de obicei utilizate pentru inițializarea flag-ului CF într-o stare cunoscută înainte de a executa o instrucțiune care folosește acest *flag* într-o operație. De asemenea, sunt utilizate în combinație cu instrucțiunile de rotire prin *carry* (RCL și RCR).

Instrucțiunile STD (Set Direction Flag) și CLD (Clear Direction Flag) permit ca flag-ul DF din registrul RFLAGS să fie modificat direct. *Flag*-ul DF determină direcția în care regiștri de index RSI și RDI sunt actualizați la executarea instrucțiunilor de procesare a șirurilor (a se vedea secțiunea 4.10). Dacă steagul DF este 0, regiștrii de index sunt incrementați după fiecare iterație a unei instrucțiuni cu șiruri, iar dacă acesta este setat pe 1, regiștrii sunt decrementați.

4.12.2. Instrucțiunea NOP

Instrucțiunea NOP (No Operation) nu face nimic, cu excepția pierderii unui ciclu de procesare și ocupă câțiva octeți în memoria de program. Programatorii folosesc adesea această instrucțiune pentru a rezerva locații în memoria de program, pentru depanare sau pentru alinierea

instrucțiunilor în memoria de program.

Instrucțiunea NOP are câteva forme:

```
nop
nop     regx
nop     memx                                    ; X = 16,32
```

Prima formă a instrucțiunii NOP nu este de fapt o instrucțiune unică, este doar un sinonim pentru „xchg eax, eax”. Această formă ocupă în memoria de program un singur octet.

Procesoarele moderne suportă și o variantă cu operand pentru intrucțiunea NOP. Acest lucru poate fi verificat prin instrucțiunea CPUID pentru EAX=01h, biții 11:8 ai valorii returnate trebuie să fie 0110b sau 1111b (a se vedea secțiunea 4.12.3).

Formele cu operand registru de uz general sau locație de memorie pe 16 sau 32 biți nu modifică conținutul registrului respectiv nu va efectua nici o operație cu memoria. Forma operand de memorie a instrucțiunii permite crearea unei secvențe de octeți de program care să execute o non-operație ca o singură instrucțiune.

Pentru situațiile în care sunt necesare NOP-uri cu mai mulți octeți, operațiile recomandate de Intel sunt cele din Tabelul 4.12 [7].

Tabelul 4.12: Operațiile recomandare pentru instrucțiunile NOP de diferite dimensiuni.

Dimensiune	Instrucțiunea	Codificarea instrucțiunii
1 octet	`nop`	`90h`
2 octeți	`66h nop`	`66 90h`
3 octeți	`nop dword ptr [eax]`	`0f 1f 00h`
4 octeți	`nop dword ptr [eax + 00h]`	`0f 1f 40 00h`
5 octeți	`nop dword ptr [eax + eax*1 + 00h]`	`0f 1f 44 00 00h`
6 octeți	`66h nop dword ptr [eax + eax*1 + 00h]`	`66 0f 1f 44 00 00h`
7 octeți	`nop dword ptr [eax + 00000000h]`	`0f 1f 80 00 00 00 00h`
8 octeți	`nop dword ptr [eax + eax*1 + 00000000h]`	`0f 1f 84 00 00 00 00 00h`
9 octeți	`66h nop dword ptr [eax + eax*1 + 00000000h]`	`66 0f 1f 84 00 00 00 00 00h`

În tabelul 4.12 instrucțiunile de dimensiune 2, 6 și 9 au prefixul 66h. Prefixul 66h (Operand Override) modifică dimensiunea implicită a operanzilor unei instrucțiuni (de exemplu: de la 16 biți la 32 biți și invers).

4.12.3. Instrucțiunea CPUID

În arhitectura x86, instrucțiunea CPUID (CPU Identification) este o instrucțiune suplimentară care permite aplicațiilor software să obțină detalii ale procesorului. Această instrucțiune a fost introdus de Intel în 1993 începând cu procesoarele Pentium și 486SL.

Flag-ul ID (bitul 21) din registrul RFLAGS indică suport pentru instrucțiunea CPUID. Posibilitatea modificării *flag*-ului ID indică faptul că procesorul suportă instrucțiunea CPUID.

În limbajul de asamblare, instrucțiunea CPUID nu are parametri, deoarece CPUID folosește implicit registrul EAX pentru a determina categoria principală de informații returnate. Instrucțiunea CPUID returnează informațiile de identificare și caracteristicile procesorului în regiștrii EAX, EBX, ECX și EDX. La procesoarele x86-64 pe 64 biți valorile returnate sunt doar în regiștri pe 32 biți, jumătatea superioară a regiștrilor RAX, RBX, RCX și RDX va fi 0. Ieșirea

instrucțiunii depinde de conținutul registrului EAX la executare (în unele cazuri și de ECX).

Astfel pentru început registrul EAX se încarcă cu valoarea 00h și apelul instrucțiunii CPUID va returna în EAX valoarea maximă de intrare pentru informațiile de bază ale procesorului și șirul de identificare al producătorului.

```
mov    eax, 00h          ; sau xor eax, eax
cpuid
```

Secvența de mai sus va returna șirul de identificare a producătorului procesorului – un șir ASCII de douăsprezece caractere stocat în EBX, EDX, ECX (în această ordine). Valoarea maximă acceptată ca parametru pentru funcțiile de bază (cea mai mare valoare la care EAX poate fi setat înainte de a apela CPUID) este returnat în EAX. Dacă procesorul este produs de Intel în regiștrii EBX, ECX și EDX se vor găsi valorile: "Genu", "ntel" respectiv "ineI". Șirul obținut în urma reordonării și concatenării acestor valori va fi: "GenuineIntel". Dacă procesorul ar fi produs de AMD, șirul returnat ar fi: "AuthenticAMD". Dacă, de exemplu, aplicația rulează sub o mașină virtuală VMware șirul returnat va fi "VmwareVMware".

Pentru a obține informații despre funcțiile extinse, CPUID trebuie apelat cu cel mai semnificativ bit al registrului EAX setat pe 1. Pentru a determina valoarea maximă a parametrului de apelare al funcțiilor extinse, CPUID se va apela cu EAX având valoarea 80000000h.

În Tabelul 4.13 se pot observa câteva dintre funcțiile instrucțiunii CPUID.

Tabelul 4.13: Funcții CPUID.

Parametru(i)	Funcție
EAX=0	Highest Function Parameter and Manufacturer ID
EAX=1	Processor Info and Feature Bits
EAX=2	Cache and TLB Descriptor information
EAX=3	Processor Serial Number
EAX=4	Intel thread/core and cache topology
EAX=Bh	
EAX=7, ECX=0	Extended Features
EAX=7, ECX=1	Extended Features
EAX=80000000h	Get Highest Extended Function Implemented
EAX=80000001h	Extended Processor Info and Feature Bits
EAX=80000002h÷80000004h	Processor Brand String
EAX=80000005	L1 Cache and TLB Identifiers
EAX=80000006	Extended L2 Cache Features
EAX=80000007h	Advanced Power Management Information
EAX=80000008h	Virtual and Physical address Sizes
EAX=8FFFFFFFh	AMD Easter Egg

4.12.4. Instrucțiunea RDTSC și RDTSCP

Contorul ciclilor procesorului (TSC – Time Stamp Counter) este un registru pe 64 biți

prezent în toate procesoarele x86 începând cu Pentium. Acesta contorizează numărul de cicluri de la resetare. TSC a fost inițial o modalitate excelentă pentru ca un program să obțină informații despre temporizarea procesorului. Odată cu apariția procesoarelor *multi-core/hyper-threaded*, a sistemelor cu mai multe procesoare și a sistemelor de operare hibernante, TSC nu mai poate fi utilizat pentru a furniza rezultate exacte – cu excepția cazului în care se acordă mare atenție pentru a corecta eventualele defecte: frecvența procesorului și dacă toate nucleele (procesoare) au valori identice în regiștrii lor de păstrare a timpului. Nu există nici o garanție că contoarele de timp pentru mai multe procesoare de pe o singură placă de bază vor fi sincronizate. Prin urmare, un program poate obține rezultate fiabile doar limitându-se să funcționeze pe un anumit procesor. Chiar și atunci, viteza procesorului se poate modifica din cauza măsurilor de economisire a energiei (Power Saving) luate de sistemul de operare sau BIOS. Sistemul poate chiar intra în starea de hibernare și ulterior reluat, resetând TSC. În aceste ultime cazuri, pentru a rămâne relevant, programul trebuie să recalibreze contorul periodic.

Instrucțiunea `RDTSC` (Read Time-Stamp Counter) citește valoarea actuală a contorului TSC în perechea de regiștri `EDX:EAX`. Registrul `EDX` este încărcat cu cei mai semnificativi 32 biți, iar registrul `EAX` este încărcat cu cei mai nesemnificativi 32 biți ai contorului (cei 32 biți mai semnificativi din regiștrii `RAX` și `RDX` sunt resetați la valoarea `0`). Instrucțiunea `RDTSC` nu are operanzi.

Flag-ul `TSD` (Time Stamp Disable) din registrul sistem `CR4` restricționează utilizarea instrucțiunii `RDTSC` după cum urmează: când *flag*-ul este resetat pe valoarea `0`, instrucțiunea `RDTSC` poate fi executată la orice nivel de privilegiu; atunci când este setat pe `1`, instrucțiunea poate fi executată doar la nivelul 0 de privilegiu.

Procesoarele bazate pe micro-arhitectura Intel cu numele de cod de Nehalem furnizează un registru auxiliar TSC, TSC_AUX, care este proiectat pentru a fi utilizat împreună cu TSC. TSC_AUX furnizează un câmp pe 32 biți, inițializat de *software* privilegiat cu o valoare de semnătură (de exemplu, un ID de procesor logic). TSC_AUX este utilizat împreună cu TSC pentru a a permite *software*-ului să citească atât cei 64 biți ai TSC și cât și valoarea semnăturii pe 32 biți din TSC_AUX cu instrucțiunea `RDTSCP` într-o operație atomică.

Similară cu instrucțiunea `RDTSC`, `RDTSCP` returnează cocntotul de cicli ai procesorului pe 64 biți în `EDX:EAX` și suplimentar valoarea de semnătură TSC_AUX pe 32 biți în `ECX`. Atomicitatea `RDTSCP` asigură că nu se poate schimba contextul între citirile valorilor TSC și TSC_AUX. Posibilitatea utilizării instrucțiunii `RDTSCP` este indicată de valoarea bitului al `27`-lea al valorii (pe 32 biți) registrului `EDX` returnat de apelul instrucțiunii `CPUID` cu funcția `EAX=80000001h (CPUID.80000001H:EDX[27])`.

Procesoarele moderne pot avea o versiune îmbunătățită a contorului `TSC` numit `TSC` invariabil (invariant TSC). Suportul procesoarelor pentru TSC invariant este indicat de bitul 8 al valorii returnate în registrul `EDX` în urma apelului funcției extinse `80000007h` a instrucțiunii `CPUID` (`CPUID.80000007H:EDX[8]`). TSC invariant va rula într-un ritm constant în toate stările procosorului. Citirile TSC sunt mult mai eficiente față de citirile *timere*-lor ACPI (Advanced Configuration and Power Interface) sau HPET (High Precision Event Timer).