

Introducere

Un calculator numeric este constituit dintr-un ansamblu de resurse fizice (hardware) și de programe de sistem (software de sistem) care asigură prelucrarea automată a informațiilor, în conformitate cu algoritmi specificați de utilizator prin programele de aplicații (software utilizator).

Arhitectura calculatorului cuprinde două componente principale:

- a) arhitectura setului de instrucțiuni (ASI);
- b) implementarea mașinii, cu cele două sub - componente:
 - organizare;
 - hardware.

Arhitectura setului de instrucțiuni este ceea ce trebuie să știe un programator pentru a scrie programe în limbaj de asamblare, respectiv pentru a concepe și construi un program de tip compilator, sau rutine destinate sistemului de operare.

Termenul organizare include aspectele de nivel înalt ale unui proiect de calculator, ca de exemplu structura internă a UCP (unitatea centrală de procesare, microprocesor), structura și organizarea magistrelor, organizarea sistemului de memorie. Noțiunea de hardware (resurse fizice) e utilizată pentru a ne referi la aspectele specifice ale implementării calculatorului. Acestea includ proiectul logic de detaliu și tehnologia de realizare a mașinii de calcul.

1. Reprezentarea funcțională a unui calculator

Un calculator poate fi descris atât sub aspect funcțional cât și structural. Circuitele electronice ale unui calculator recunosc și execută doar un set limitat de *instrucțiuni elementare*, codificate în formă binară. Aceste instrucțiuni sunt doar succesiuni de biți (1 și 0) pe care procesorul le înțelege - decodifică și le execută. Indiferent de tipul de mașină, instrucțiunile recunoscute sunt rareori mai complicate decât [Tanenbaum]:

- adună două numere;
- verifică dacă un număr este egal cu zero;
- copiază date dintr-o zonă a memoriei calculatorului în altă zonă.

De exemplu, la procesoarele Intel din seria 80x86 codul binar al instrucțiunii următoare:

0000 0100 0000 0110

comandă adunarea conținutului unui registru intern de 8 biți (numit registrul al) cu valoarea imediată 6. Adesea, când utilizatorul este obligat să lucreze cu valori numerice binare, se folosește reprezentarea în hexazecimal, care este mai compactă și mai ușor de citit. Codul, în hexazecimal, al instrucțiunii corespunzătoare succesiunii binare de mai sus este 04 06 hex. Și acest mod de scriere a instrucțiunilor este însă complicat pentru programator. În assembler această instrucțiune se descrie **mov al,6**.

Scrierea instrucțiunilor se poate realiza, la început, în limbaj natural, transpunându-se apoi într-un limbaj artificial, numit *limbaj de programare*.

Limbajul de programare reprezintă un set de instrucțiuni, împreună cu regulile de organizare ale acestora într-un program. Totalitatea regulilor de scriere ale instrucțiunilor reprezintă *sintaxa* limbajului, iar totalitatea regulilor prin care se asociază o semnificație instrucțiunilor reprezintă *semantica* limbajului.

Calculatorul poate executa numai instrucțiuni exprimate intern sub forma unor șiruri de cifre binare. Programele în care instrucțiunile sunt scrise sub această formă se numesc programe în *limbaj mașină*. Limbajul mașină este caracteristic fiecărui tip de calculator. Scrierea și introducerea programelor sub această formă este dificilă. Pentru simplificarea scrierii programelor, au fost create limbaje în care fiecărui cod de instrucțiune i s-a atașat un *nume mnemonic*. Acest mod de reprezentare a instrucțiunilor mașină se numește *limbaj simbolic* sau *limbaj de asamblare*. Acest limbaj permite utilizatorului să realizeze codificări simbolice ale instrucțiunilor și ale adreselor de memorie.

Pentru a se executa un program în limbaj de asamblare, acesta trebuie translatat în prealabil în limbaj mașină, printr-un proces numit *asamblare*. Programul care se translatează se numește *program sursă*, iar cel rezultat în urma translatarei se numește *program obiect*. Operația de traducere a

programului sursă în program obiect este executată de un program special numit *asamblor*.

O instrucțiune în limbaj de asamblare corespunde unei instrucțiuni în limbaj mașină, deosebirea dintre ele constând numai în modul de reprezentare. Deci, fiecare instrucțiune în limbaj de asamblare este translatată într-o singură instrucțiune în limbaj mașină.

Un limbaj de asamblare este specific unui calculator, astfel încât trebuie să se cunoască instrucțiunile și organizarea internă a acelui calculator. Din acest motiv, s-au elaborat limbaje pentru programe care se pot executa pe orice calculator. Acestea sunt limbaje orientate pe probleme sau *limbaje de nivel înalt*, spre deosebire de limbajele de asamblare, care sunt limbaje orientate pe calculator sau *limbaje de nivel scăzut*. Asemenea limbaje de nivel înalt sunt Pascal, C, LISP, PROLOG etc.

Unei instrucțiuni într-un limbaj de nivel înalt îi corespunde o succesiune de instrucțiuni în limbaj mașină. Traducerea în limbajul mașină se poate realiza cu ajutorul unui *compiler*, care generează din programul sursă un *program executabil*, acesta fiind executat după ce întregul program a fost compilat.

O altă posibilitate este utilizarea unui *interpretor*, care translatează fiecare instrucțiune în limbajul de nivel înalt într-o succesiune de instrucțiuni mașină, acestea fiind executate imediat. În acest caz nu se generează un program executabil. Viteza de execuție este însă redusă, deoarece fiecare instrucțiune trebuie interpretată chiar dacă ea este executată în mod repetat.

Uneori este mai convenabil să se considere că există un calculator ipotetic sau o *mașină virtuală*, a cărei limbaj mașină este un anumit limbaj de nivel înalt. Un asemenea calculator ar executa direct instrucțiunile limbajului de nivel înalt, fără a fi necesară utilizarea unui translator (compiler) sau interpretor. Chiar dacă implementarea unei mașini virtuale care să lucreze direct cu un limbaj de nivel înalt ar fi prea costisitoare, se pot scrie programe pentru această mașină, deoarece aceste programe pot fi translatate sau interpretate cu un program care poate fi executat direct de calculatorul existent.

Modelul unui calculator numeric

Un model posibil al unui calculator numeric modern reprezintă o ierarhie de mașini virtuale pe mai multe nivele (Figura 1.1).

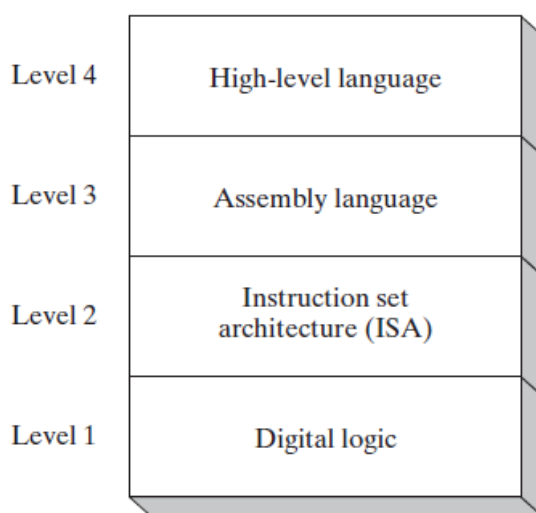


Figura 1.1 - Ierarhia de nivele a unui calculator modern

Nivelul 1, numit *nivelul logicii digitale*, este reprezentat de componentele hardware ale calculatorului (mașina fizică). Circuitele acestui nivel execută instrucțiunile mașină ale nivelului 2. Elementele de bază ale acestor circuite sunt *porțile logice*, fiecare poartă fiind formată la rândul ei dintr-un număr de tranzistoare. O poartă logică are una sau mai multe intrări digitale (semnale reprezentând 0 logic sau 1 logic), și are ca ieșire o funcție simplă a acestor intrări, de exemplu ȘI logic, SAU logic.

Nivelul 2 reprezintă setul de instrucțiuni ale unei arhitecturi concrete de microprocesor. Producătorii de microprocesoare proiectează microprocesoarele sale ca să recunoască un anumit set de instrucțiuni pentru a efectua operațiuni de bază, cum ar fi copierea, adunarea, sau multiplicarea. Acest set de instrucțiuni este menționat ca limbaj mașină. Fiecare instrucțiune din limbajul mașină este executat fie direct prin hardware-ul calculatorului sau de un microprogram încorporat în cipul microprocesorului.

De menționat că la anumite calculatoare nivelul de microprogram lipsește. La aceste calculatoare, instrucțiunile mașinii convenționale sunt executate direct de circuitele electronice ale nivelului 1.

Nivelul 3 este *nivelul limbajului de asamblare*. Limbaj de asamblare, care apare la nivelul 3, folosește mnemonice scurte, cum ar fi ADD, SUB, și MOV, care sunt ușor de compilate (asamblate) la nivelul ISA.

Nivelul 4 constă din limbajele destinate programatorilor de aplicație, fiind numit *nivelul limbajelor de nivel înalt*. Programele scrise în aceste limbaje sunt translatate în o mulțime de instrucțiuni a limbajului nivelului 3 cu ajutorul compilatoarelor sau interpretoarelor.

Fiecare nivel reprezintă o abstractizare distinctă, cu diferite obiecte și operații. Setul tipurilor de date, a operațiilor și facilităților fiecărui nivel reprezintă arhitectura nivelului respectiv. Arhitectura tratează acele aspecte care sunt vizibile utilizatorului nivelului respectiv, ca de exemplu dimensiunea memoriei disponibile. Aspectele de implementare, ca de exemplu tehnologia utilizată pentru implementarea memoriei, nu fac parte din arhitectură. *Arhitectura calculatorului* reprezintă studiul proiectării acelor părți ale unui sistem de calcul care sunt vizibile pentru programatori.

2 Registrele microprocesoarelor

2.1 Noțiuni generale

În Assembler, calculatorul este văzut la nivelul hardware: adrese fizice de memorie, registre, întreruperi etc. Sunt necesare unele noțiuni pregătitoare.

Unitatea de bază a informației memorate în calculator este **bitul**. Un bit reprezintă o cifră binară (de aici și numele, care e o prescurtare de la binary digit), deci poate avea valorile 0 sau 1. Modelul hardware corespunzător este acela de **bistabil**. Un bistabil este deci un circuit electronic cu două stări stabile, codificate 0 și 1, capabil să memoreze un bit de informație.

Un grup de bistabili formează un **registru**. De exemplu, 8 bistabili formează un registru de 8 biți. Informația care se poate memora într-un asemenea registru poate fi codificată în binar, de la valoarea 00000000 (toți biții egali cu 0), până la valoarea 11111111 (toți biții egali cu 1). Este ușor de văzut că numărul combinațiilor care pot fi memorate este 256 (2 la puterea a 8-a). În general, un registru de n biți va putea memora 2^n combinații distincte. Aceste combinații se numesc **octeți** sau **bytes** (dacă $n = 8$), respectiv **cuvinte** (dacă $n = 16, 32$ etc).

2.2 Registrele microprocesoarelor x86-64

Fiecare program la execuție obține anumite resurse ale microprocesorului. Aceste resurse (registre) sunt necesare pentru executarea și păstrarea în memorie a instrucțiunilor și datelor programului, a informației despre starea curentă a programului și a microprocesorului.

Microprocesoarele pe 32 biți funcționează în diferite moduri, ce determină mecanismele de protecție și de adresare a memoriei: modul real 8086 (pe 16 biți), modul virtual 8086 (V8086), modul protejat pe 32 biți (inclusiv protejat pe 16 biți). Modul de funcționare a microprocesorului este impus de sistemul de operare (SO) în conformitate cu modul definit de aplicații (task-uri).

În microprocesoarele pe 64 biți au fost introduse noi moduri de funcționare:

- Modul pe 64 biți (64-bit mode) – acest mod susține adresarea virtuală pe 64 biți și extensiile registrelor pe 64 biți. În acest mod este folosit numai *modelul plat de memorie* (un segment comun pentru cod, date și stivă).
- Modul de compatibilitate (compatibility mode) permite SO să execute aplicații pe 32 și 16 biți. Pentru aplicații microprocesorul reprezintă un microprocesor pe 32 biți cu toate atributele modului protejat, cu mecanismele de segmentare și paginare.

Microprocesoarele pe 64 biți reprezintă seturi de registre disponibile programatorilor (figura 2.1).

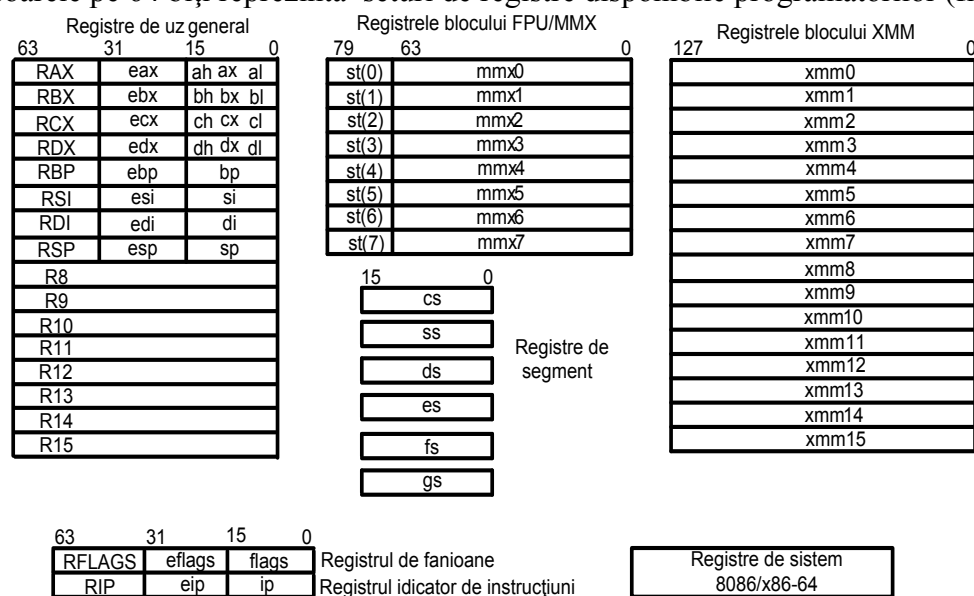


Figura 2.1 Registrele microprocesoarelor x86-64

Structura și destinația regiștrilor setului FPU/MMX, XMM, de sistem vor fi prezentate în capitolul „Modelul program ale microprocesoarelor”.

2.2.1 Regiștrii de uz general

În figura 2.2 sunt prezentați regiștrii de uz general pe 64 biți.

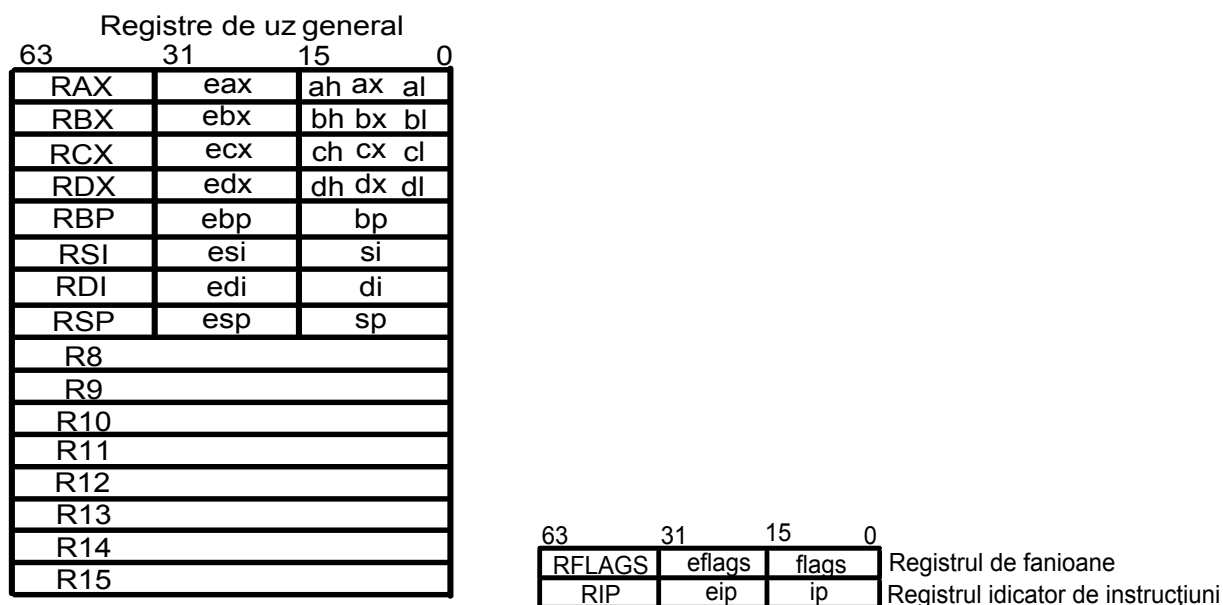


Figura 2.2 - Regiștrii de uz general pe 64 biți

Regiștrii pe 64 biți sunt indicați cu prefixul (REX). Adresarea la fiecare din 16 registre se petrece ca la un registru pe 64-, 32-, 16- sau 8 biți (se folosesc numai biții inferiori). Structura și destinația registrelor va fi detaliată în subcapitolele următoare, în descrierea modurilor de compatibilitate.

2.3 Modul de compatibilitate pe 16 biți

Pentru modul de compatibilitate pe 16 biți (modul 8086), sunt utilizați numai părțile inferioare pe 16 biți (figura 2.3) ai registrelor microprocesoarelor pe 64 (32) biți, setul de registre este următorul:

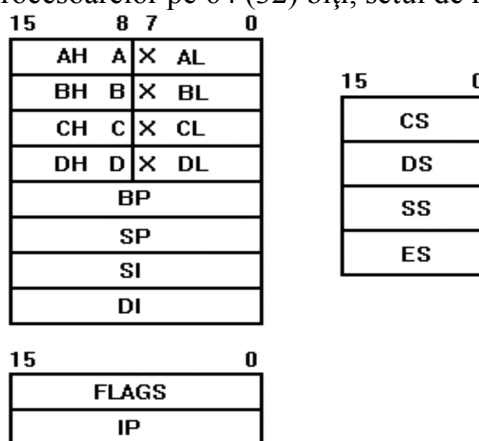


Figura 2.3

Toate registrele sunt de 16 biți. O serie de registre (AX, BX, CX, DX) sunt disponibile și la nivel de octet, părțile mai semnificative fiind AH, BH, CH și DH, iar cele mai puțin semnificative, AL, BL, CL și DL. Denumirile registrelor sunt:

- AX - registru acumulator
- BX - registru de bază general
- CX - registru contor
- DX - registru de date

- BP - registru de bază pentru stivă (base pointer)
- SP - registru indicator de stivă (stack pointer)
- SI - registru index sursă
- DI - registru index destinație

Registru notat FLAGS cuprinde flagurile (biți indicatori) procesorului, sau bistabililor de condiție, iar registru IP (instruction pointer) este registru de instrucțiuni.

Registrele de date (AX, BX, CX, DX) pot fi folosite pentru memorarea datelor, însă unele din acestea sunt folosite special pentru alte scopuri. De exemplu registru CX este folosit ca contor în instrucțiunea LOOP de ciclare, care îl modifică implicit și-i testează valoarea, registru DX este folosit în instrucțiunile de împărțire și înmulțire, iar registru BX este folosit pentru adresarea datelor (operandilor).

În modul 8086 unui program scris în limbaj de asamblare se alocă patru zone de lucru în memorie: o zonă pentru date, o zonă pentru instrucțiuni, o zonă specială pentru stivă și o zonă suplimentară pentru date. Fiecare din aceste zone pot avea până la 64K octeți și poartă denumirea de segment. Astfel există segmentul de date (Data Segment), segmentul de instrucțiuni, cod (Code Segment), segmentul de stivă (Stack Segment) și segmentul suplimentar de date (Extra Segment).

Este posibil ca cele 4 segmente să fie suprapuse total sau parțial. Adresele de început ale acestor segmente se află în 4 registre segment.

Denumirile registrelor de segment sunt:

- CS - registru de segment de cod (code segment);
- DS - registru de segment de date (data segment);
- SS - registru de segment de stivă (stack segment);
- ES - registru de segment de date suplimentar (extra segment).

Se observă că denumirile registrelor de segment corespund zonelor principale ale unui program executabil. Astfel, perechea de registre (CS:IP) va indica totdeauna adresa următoarei instrucțiuni care se va executa, iar perechea (SS:SP) indică totdeauna adresa vârfului stivei. Registrele DS și ES conțin adresele segmentelor de date și sunt folosite pentru a accesa date.

Dacă segmentul de date începe de la locația de memorie 1234h atunci DS va conține valoarea 1234h. Există instrucțiuni pentru încărcarea adreselor de memorie în registrele segment.

Registrele pointer (SP și BP) se folosesc pentru calculul offsetului (distanței față de începutul unui segment) din cadrul segmentului. Cele două registre pointer sunt: pointerul de stivă SP (Stack Pointer) și pointerul de bază (Base Pointer). SP și BP sunt registre de 16 biți.

Registru SP reține adresa efectivă (offsetul) a vârfului stivei (figura 2.4). Adresa fizică a vârfului stivei SS:SP este dată de perechea de registre SS și SP, registru SS conține adresa de început al segmentului de stivă iar SP conține offsetul din acest registru (adică distanța în octeți de la începutul registrului de stivă):

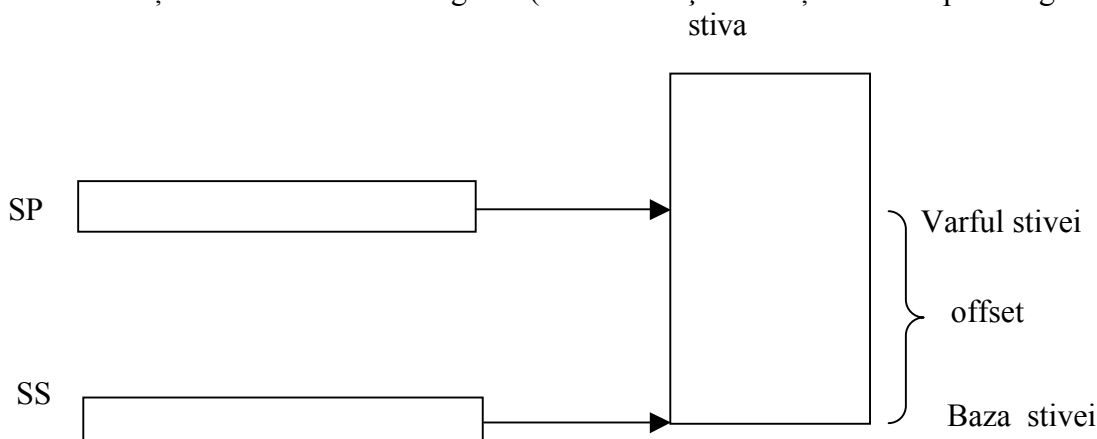


Figura 2.4

Registru BP este folosit la calculul offset-ului din interiorul unui segment. De exemplu poate fi folosit ca pointer al unei stive proprii, dar nu a procesorului. Este folosit în principal pentru adresarea bazată indexată a datelor.

Registrele de index, de 16 biți sunt: SI (Source Index) și DI (Destination Index). Registrele de index sunt folosite pentru accesul la elementele unui tablou sau a unei tabele. Aceste registre sunt folosite îndeosebi în prelucrarea șirurilor de caractere.

Registru de instrucțiuni (Instruction Pointer) conține offsetul curent în segmentul de cod. Adică

adresa efectivă a următoarei instrucțiuni de executat din segmentul de cod curent. După executarea instrucțiunii curente, microprocesorul preia din IP adresa următoarei instrucțiuni de executat și incrementează corespunzător valoarea lui IP, cu numărul de octeți ai codului instrucțiunii ce va fi executată. Uneori acest registru se numește numărător de program.

Registru de flag-uri (fanioane) (bistabili de condiție) al modului 8086 are configurația din figura 2.5. O serie de flag-uri sunt flag-uri de stare: acestea sunt poziționate la 0 sau la 1 ca urmare a unor operații aritmetice sau logice, conțin informații despre ultima instrucțiune executată. Celelalte flag-uri controlează anumite operații ale procesorului.

Din cei 16 biți ai registrului sunt folosiți 9 biți: 0, 2, 4, 6 – 11.

Aproape toate instrucțiunile limbajului de asamblare afectează biții de stare.

Semnificația flag-urilor (biților) este următoarea:

- CF (Carry Flag, bistabil de transport) - semnifică un transport sau un împrumut din/în bitul cel mai semnificativ al rezultatului, de exemplu la operații de adunare sau de scădere.

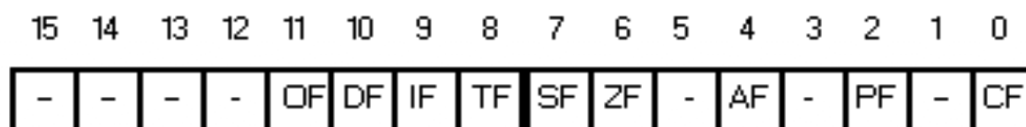


Figura 2.5 - Registrul de flag-uri al procesorului 8086

- PF (Parity Flag, flag de paritate) - este poziționat în așa fel încât numărul de biți egali cu 1 din octetul cel mai puțin semnificativ al rezultatului, împreună cu flag-ul PF, să fie impar; altfel formulat, suma modulo 2 (XOR) a tuturor biților din octetul c.m.p.s. și a lui PF să fie 1.
- AF (Auxiliary Carry Flag, bistabil de transport auxiliar) - indică un transport sau un împrumut din/în bitul 4 al rezultatului.
- ZF (Zero Flag, bistabil de zero) - este poziționat la 1 dacă rezultatul operației este 0.
- SF (Sign Flag, bistabil de semn) - este poziționat la 1 dacă b.c.m.s. al rezultatului (bitul de semn) este 1.
- OF (Overflow Flag, bistabil de depășire) - este poziționat la 1 dacă operația a condus la o depășire de domeniu a rezultatului (la operații cu sau fără semn).
- TF (Trap Flag, bistabil de urmărire) - dacă este poziționat la 1, se forțează o întrerupere, pe un nivel predefinit, la execuția fiecărei instrucțiuni; acest fapt este util în programele de depanare, în care este posibilă rularea pas cu pas a unui program.
- IF (Interrupt Flag, bistabil de întreruperi) - dacă este poziționat la 1, procesorul ia în considerație întreruperile hardware externe; altfel, acestea sunt ignorate.
- DF (Direction Flag, bistabil de direcție) - precizează sensul (crescător sau descrescător) de variație a adreselor la operațiile cu șiruri de octeți sau de cuvinte.

Flag-urile CF, PF, AF, ZF, SF și OF sunt numite flag-uri de stare (aritmetice). Flag-urile TF, IF și DF sunt numite flag-uri de control.

2.3.1 Formarea adresei fizice

Procesorul 8086 dispune de adrese pe 20 de biți, fiind capabil să adreseze 1 megaoctet de memorie (2^{20}). Se pune problema cum se formează adresa fizică pe 20 de biți (deci pe 5 cifre hexa), deoarece toate registrele procesorului sunt de 16 biți, putând codifica adrese în domeniul 0000...0FFFFH (pe 4 cifre hexa), deci într-un spațiu de maxim 64 KO.

Memoria unui sistem cu procesor 8086 este divizată în segmente. Un segment este o zonă continuă de memorie, de lungime maximă de 64 KO, care începe la o adresă fizică multiplu de 4. Acest fapt înseamnă că ultima cifră hexa a adresei de început a unui segment este totdeauna 0. Ca atare, această cifră se poate omite și adresa de segment se poate reprezenta tot pe 16 biți. Adresele de început ale segmentelor se vor găsi întotdeauna în unul din cele 4 registre de segment.

Adresarea în interiorul unui segment se realizează printr-un deplasament (offset) relativ la începutul segmentului. Deoarece un segment nu poate depăși 64 KO, deplasamentul se poate memora tot pe 16 biți. Deplasamentul poate fi o constantă sau conținutul unui registru care permite adresarea memoriei.

În concluzie, pentru adresarea unui octet de memorie, se folosesc două entități pe 16 biți: o adresă de segment (conținută obligatoriu într-un registru de segment) și un deplasament. Deoarece ambele entități sunt pe 16 biți, se vorbește de adrese (sau pointeri) de 32 de biți, deși adresa fizică este doar pe 20 de biți.

Formarea adresei fizice (pe 20 de biți) este realizată automat (prin hardware) de către o componentă a procesorului, conform Figurii 2.6.

Concret, adresa fizică se obține prin deplasarea adresei de segment cu 4 biți la stânga și prin adunarea deplasamentului. Pentru specificarea unei adrese complete (de 32 de biți), se folosește notația (segment:offset) sau (registru_segment:offset). De exemplu, putem specifica o adresă prin (18A3:5B27) sau prin (DS:5B27).

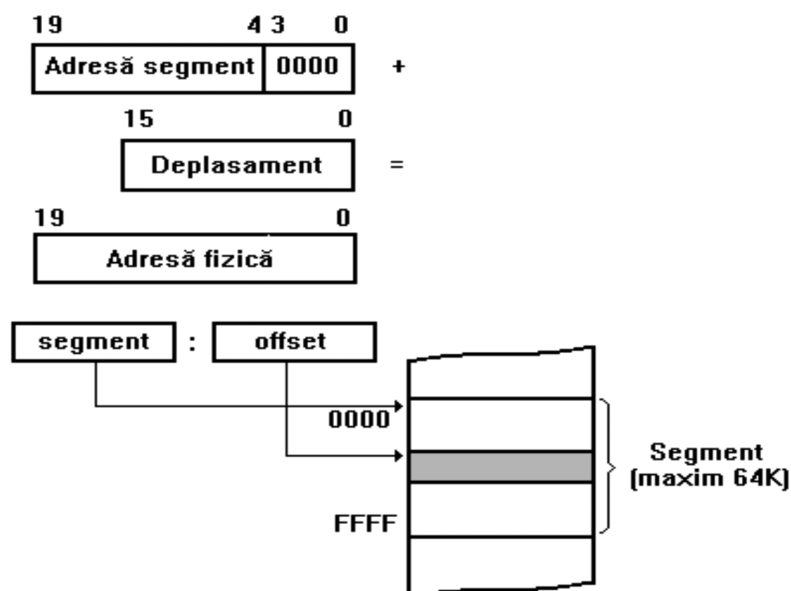


Figura 2.6 - Formarea adresei fizice

Trebuie remarcat faptul că asocierea (segment:offset) - adresă fizică nu este biunivocă, deoarece la o aceeași adresă fizică pot să corespundă mai multe perechi (segment:offset). De exemplu, perechile (18A3:5B27) și (18A2:5B37) reprezintă aceeași adresă fizică. În situația în care deplasamentul este redus la minim, adică în domeniul 0...F, corespondența devine biunivocă.

O adresă completă de 32 de biți este memorată cu offsetul la adrese mici și cu adresa de segment la adrese mari. Adresele complete se pot obține cu directiva DD (Define Double-Word).

2.3.2 Definirea segmentelor. Structura programelor

Segmentele logice conțin cele trei componente ale unui program: cod, date și stivă.

Pentru a scrie un program ASM (în modul 8086), se folosesc directivele simplificate de definire a segmentelor. Acestea sunt:

- **.model small** (precizează un model de memorie)
- **.stack n** (definire de segment de stivă)
- **.data** (definire de segment de date)
- **.code** (definire de segment de cod)
- **end etichetă** (sfârșit logic al programului)

În principiu, segmentul de cod va cuprinde programul executabil (instrucțiuni), iar segmentul de date va cuprinde date definite de utilizator. Segmentul de stivă nu este folosit explicit. Comentariile se scriu folosind simbolul ;. Ceea ce urmează după ; până la sfârșitul liniei curente, este considerat comentariu.

Structura programului este:

```
.MODEL small
.STACK 512
.DATA
```

definirea datelor

declarare si definire proceduri

```
start: mov ax,@data  
mov ds,ax
```

program principal

```
end start
```

Directiva **.stack** alocă o zonă având lungimea **n (.stack n)**, zonă fiind definită ca stivă (ex: `.stack 200h` va alocă un segment de lungime 512 octeți). Directiva rezervă (nu și inițializează) zona dedicată stivei. Acțiunea de inițializare este opțională. Putem scrie și `.stack 512`.

Directiva **.code** precede segmentul de program. Încărcarea acestui segment este realizată automat de către DOS. În schimb registrul **ds** va trebui încărcat de către programator.

Simbolul **@data** va primi adresa segmentului de date, abia după momentul editării legăturilor. Date pot exista și în cadrul segmentului de cod. Registrele de segment nu sunt niciodată încărcate cu valori absolute. Se lasă în seama sistemului de operare sarcina amplasării în memorie a segmentelor. Ordinea este asigurată tot de către componentele sistemului de operare.

Directiva **.model** definește modul de dispunere în memoria RAM a segmentelor care alcătuiesc un program. Sistemul DOS admite 6 modele.

Cele mai des utilizate modele de memorie sunt tiny și small (pot fi și medium, large sau huge). Semnificația acestor tipuri este:

- **tiny** - toate segmentele (date, cod, stivă) se pot genera într-un spațiu de 64KO și formează un singur grup de segmente. Se folosește la programele de tip COM.
- **small** - datele și stiva sunt grupate într-un singur segment iar codul în alt segment. Fiecare din acestea nu depășesc 64KO.

Etichetele sunt nume simbolice de adrese (offset) ce identifică instrucțiunile. Etichetele pot fi referite în alte instrucțiuni pentru executarea salturilor în program. Dacă referirile la o etichetă sunt făcute în cadrul segmentului în care ea este definită atunci se spune ca ea are atributul **NEAR**.

Pentru declararea etichetelor în segmentul de program se utilizează operatorul **:** (ex: `START:`)

Directiva **END** marchează sfârșitul logic al unui modul de program și e obligatorie în toate modulele. Tot ce se găsește în fișierul sursă după această directivă este ignorat la asamblare. Forma generală este:

```
END [punct_de_start]
```

în care `punct_de_start` este o etichetă sau un nume de procedură care marchează punctul în care se va da controlul după încărcarea programului în memorie.

2.4 Modul de compatibilitate pe 32 biți (microprocesor de 32 biți)

Pentru modul de compatibilitate pe 32 biți, sunt utilizate numai părțile inferioare pe 32 biți ai registrelor microprocesoarelor pe 64 de biți și setul de registre de uz general este prezentat în figura 2.7. Microprocesoarele de 32 biți sunt compatibile ca arhitectură cu cele de 16 biți, prin aceea ca registrele de 16 biți se regăsesc ca subregistre ale registrelor de 32 de biți. Pentru accesarea registrelor de 32 biți a fost adăugat un set de instrucțiuni.

Registrele din figură, de exemplu `al`, `ah`, `ax`, indică registre pe 8 și 16 biți ale registrului extins `eax` pe 32 biți (prefix „e” (Extended)).

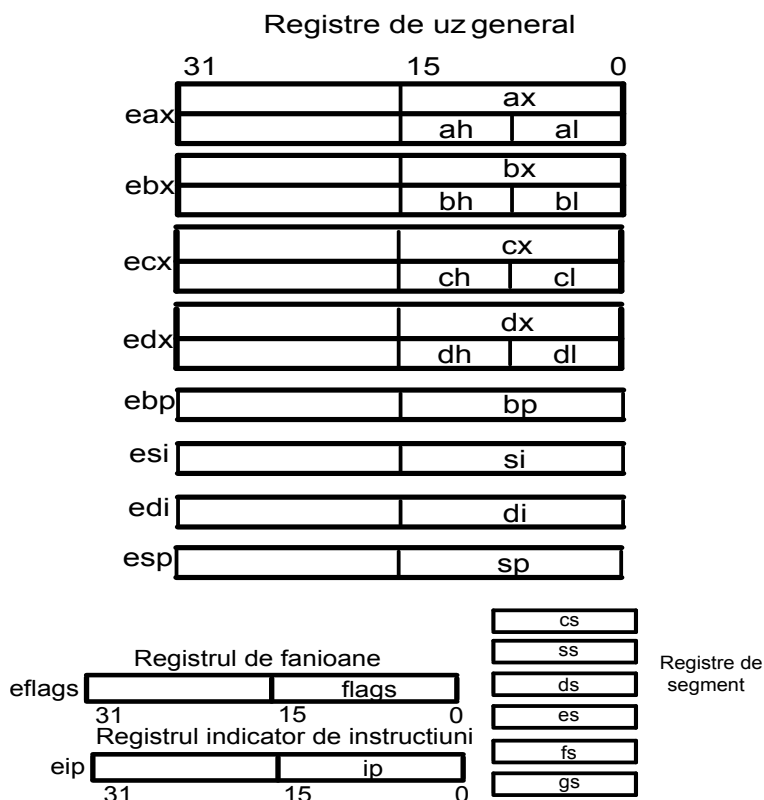


Figura 2.7 - Setul de registre de uz general

Registrele generale ax, bx, cx, dx, si, di, bp și sp de 16 biți fac parte din registrele generale de 32 biți ai microprocesoarelor de 32 biți extinse: eax, ebx, ecx, edx, esi, edi, ebp si esp. Primii 16 biți din aceste registre sunt registre generali ai microprocesoarelor de 16 biti.

Analog registrele IP si FLAGS de 16 biți sunt extinse la 32 biți în cazul registrelor EIP și ELA GS de 32 biți. Registrul FLAGS se regăsește în primii 16 biți ai registrului EFLAGS.

Registrele segment au fost păstrate de 16 biți, dar s-au adăugat două noi registre FS și GS.

Pe lângă aceste registre, microprocesoarele de 32 (64) de biți dispun de alte registre de control, de gestionare a adresei, de depanare și de test, care diferă de la un tip de procesor la altul fiind folosite în principal de programele de sistem.

Semnificația registrelor segment în cazul microprocesoarelor de 32 biți a fost modificată, ele sunt folosite ca selectoare de segment (detaliat în capitolul – memoria virtuală). În acest caz ele nu indică o adresă de segment, ci un descriptor de segment care precizează adresa de bază a segmentului, dimensiunea acestuia și drepturile de acces asociate acestuia. Astfel adresa de bază poate fi specificată pe 32 biți iar dimensiunea unui segment să fie de până la 4 GB.

2.5 Tipuri de date

Tipurile de date sunt următoarele.

Bitul. Cel mai mic element de memorare a unei informații este bitul, în care se poate memora o cifra binară, 0 sau 1.

De obicei informația de prelucrat se reprezintă pe segmente contigue de biți denumite tetrade, octeți, cuvinte, dublu cuvinte, quadwords si tenbytes.

Tetrada. Tetrada este o secvență de 4 biți, numerotați 0,1,2,3 de la dreapta la stânga, bitul 0 fiind cel mai puțin semnificativ, iar bitul 3 cel mai semnificativ:

1	0	1	1
3	2	1	0

Octetul (Byte). Octetul sau byte este un element de memorare, ce cuprinde o secvență de 8 biți. Octetul este unul dintre cele mai importante elemente (celule) de memorare adresabile. Cei 8 biți ai unui octet sunt numerotați cu 0,1,2,...7 de la dreapta la stânga:

0	1	1	0	0	0	0	1
7	6	5	4	3	2	1	0

Octetul este format din 2 tetrade, tetradă inferioara (din dreapta) conține biții 0, 1, 2, 3, iar cea superioara (din stânga) conține biții 4, 5, 6, 7 ai octetului.

Cuvântul (Word). Cuvântul este o secvență de 2 octeți, respectiv 16 biți, numerotați de la dreapta spre stânga, astfel 0, 1, 214, 15. Bitul cel mai semnificativ este bitul 15. Primul octet(inferior) din cuvânt conține biții 0, 1, 2, 3, 4, 5, 6, 7, iar al doilea octet(superior), biții 7, 8, 9, 10, 11, 12, 13, 14, 15.

1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Cuvântul poate fi reprezentat printr-un registru de 16 biți sau în doi octeți de memorie. În memorie, octetul inferior (biții 0-7) este memorat la adresa mai mică, iar octetul superior (biții 8-15) la adresa cea mai mare.

De exemplu cuvântul 4567h se reprezintă într-un registru de 16 biți sub forma 4567h, iar în memorie la adresa 1000 sub forma 6745 (octetul 67 la adresa 1000, iar octetul 45 la adresa 1001).

Dublu cuvânt (Double Word). O succesiune de 2 cuvinte (4 octeți, 32 biți), reprezintă un dublu cuvânt. Cei 32 de biți ai unui dublu cuvânt sunt numerotați de la dreapta la stânga prin 0, 1, 2,30, 31. Bitul cel mai semnificativ este bitul 31, octetul cel mai puțin semnificativ conține biții 0-7, iar cel mai semnificativ octet (octetul 4) conține biții 23-31.

Un dublu cuvânt poate fi reprezentat într-un registru de 32 biți sau pe 4 octeți consecutivi de memorie. În memorie, octetul 1-cel mai puțin semnificativ este memorat la adresa cea mai mică, iar octetul 4-cel mai semnificativ la adresa cea mai mare (în ordinea *little-endian*).

De exemplu dublul cuvânt 12 34 56 78h, aflat la offset-ul 0000, va fi memorat astfel 78 56 34 12, cu octetul 78h la offset-ul 0000, iar octetul 12h la offset-ul 0003.

0000:	78
0001:	56
0002:	34
0003:	12

Quadword. Quadword (qword) este format din 2 dublu cuvinte (4 cuvinte, respectiv 8 octeți succesivi de memorie). Cei 64 biți ai unui qword sunt numerotați de la dreapta la stânga astfel: 0, 1, 2,62, 63. Bitul cel mai semnificativ este bitul 63. În memorie octetul 1 se reprezintă la adresa cea mai mică, iar octetul 8 la adresa cea mai mare.

Tenbyte (10 octeți)

O succesiune de 10 octeți formează un tenbyte (tb). Cei 80 de biți ai elementului sunt numerotați de la dreapta la stânga cu 0, 1, 2,78, 79. În memorie octetul cel mai puțin semnificativ (biții 0-7) se reprezintă la adresa cea mai mică, iar octetul 10 (biții 73-80) la adresa cea mai mare.

2.6 Definirea datelor

În limbajele de asamblare 80x86 se poate opera cu anumite tipuri de date, recunoscute de procesor, acesta dispunând de directive (pseudoinstrucțiuni) specifice pentru definirea lor.

a) Byte (octet).

Acest tip de date ocupa 8 biți, adică un octet (byte). Informația dintr-un octet poate fi: un *întreg fără semn* cuprins între 0 și 255, un *întreg cu semn* cuprins între -128 și 127, sau un *caracter ASCII*.

Definirea datelor de tip byte se face cu ajutorul directivelor BYTE și SBYTE:

```
value1 BYTE 'A' ; character ASCII
value2 BYTE 0 ; byte fără semn
value3 BYTE 255 ; byte fără semn
value4 SBYTE -128 ; byte cu semn
value5 SBYTE +127 ; byte cu semn
value6 BYTE ? ; byte nedefinit
```

Definirea datelor de tip byte se face și cu ajutorul directivei DB (*Define Byte*):

Fie directivele:

```
alfa DB 65, 72h, 75o, 11011b, 11h+22h, 0ach
      DB -65, 'a', 'abc'
```

În memorie începând de la adresa simbolică *alfa* (offset, etichetă de date), se va genera secvența de octeți, reprezentată în hexazecimal :

41	72	3d	1b	33	ac	bf	61	61	62	63	
alfa +0	+1	+2	+3	+4						+10	

Valoarea binară 11011b va fi generată la adresa alfa+3.

Definirea șirurilor de caractere:

```
salutare1 BYTE "Good afternoon",0
greeting2 BYTE "Good night",0
```

Utilizarea operatorului DUP (duplicate):

```
BYTE 20 DUP(0) ; 20 bytes, toate încărcate cu zero
BYTE 20 DUP(?) ; 20 bytes, nedefiniți
BYTE 4 DUP("STACK") ; 20 bytes: "STACKSTACKSTACKSTACK"
```

b) WORD (cuvânt).

Un cuvânt ocupa doi octeți (16 biți) și poate fi reprezentat într-un registru de 16 biți sau în 2 octeți consecutivi de memorie. Numerotarea biților în cadrul unui cuvânt se face de la 0 la 15 (bitul 15 e bitul cel mai semnificativ al cuvântului, iar bitul 0 este bitul cel mai puțin semnificativ), numerotarea se face de la dreapta la stânga:

Informația memorată într-un cuvânt poate fi :

- un întreg pe 16 biți cu semn (bitul 15 este bitul de semn), cuprins între -2^{15} și $2^{15}-1$,
- un întreg pe 16 biți fără semn, cuprins între 0 și 2^{16}
- o adresa de memorie de 16 biți.

Reprezentarea celor 2 octeți ai cuvântului în memorie se face astfel încât octetul cel mai puțin semnificativ este memorat la adresa cea mai mică. De exemplu: dacă valoarea 2345h este memorată la adresa 2000h, atunci octetul 45h se va afla la adresa 2000h, iar octetul 23h la adresa 2001h.

Generarea datelor de tip cuvânt se poate face folosind directivele de tip WORD și SWORD:

```
word1 WORD 65535 ; întreg pe 16 biți fără semn
word2 SWORD -32768 ; întreg pe 16 biți cu semn
word3 WORD ? ; neinițializat
```

Generarea datelor de tip cuvânt se poate face și cu directiva de tip DW (*Define Word*):

Fie secvența de directive:

```
beta DW 4567h, 0bc4ah, 1110111011b, 2476o
      DW -7683, 7683, 'ab'
```

În memorie de la adresa “beta” se vor genera octeții:

67	45	4a	bc	bb	03	3e	05	fd	e1	03	e1	62	61
beta		+2		+4		+6		+8		+12			

Constanta octală 2476o este generată de la adresa beta +6.

c) **Double WORD**(dublu cuvânt)

Un dublu cuvânt ocupa 2 cuvinte sau 4 octeți (32 biți) și poate fi reprezentat în memorie pe 4 octeți consecutivi, într-o pereche de registre de 16 biți sau într-un registru de 32 biți (la procesoarele de 32 biți).

Informația memorată într-un dublu cuvânt poate fi:

- un întreg pe 32 biți, cu sau fără semn;
- un număr real în simplă precizie;
- sau o adresă fizică de memorie de 32 biți.

Generarea datelor de tip dublu cuvânt se poate face folosind directivele **DWORD** și **SDWORD**:

```
val1 DWORD 12345678h ; fără semn
val2 SDWORD -21474836 ; cu semn
val3 DWORD 20 DUP(?) ; fără semn
```

Generarea datelor de tip dublu cuvânt se face și cu directiva **DD** (*Define Double Word*):

Reprezentarea celor două cuvinte a unui dublu cuvânt de memorie se face astfel încât cuvântul cel mai puțin semnificativ este memorat la adresa cea mai mică. De exemplu dublul cuvânt 12345678 h, aflat la adresa 2000h se memorează astfel: cuvântul 5678h se memorează la adresa 2000h, iar cuvântul 1234h la adresa 2002h.

Secvența de directive :

```
val1 DD 12345678h ; fără semn
val2 DD -21474836 ; cu semn
```

d) **QUAD – WORD (8 octeți)**

Tipul Quad – word (**QWORD**) ocupa 8 octeți și este reprezentat în memorie pe 64 biți sau într-o pereche de registre de 32 biți (în cazul procesoarelor de 32 biți), sau într-un registru pe 64 biți.

Informația stocată într-un qword poate fi: un întreg cu sau fără semn pe 64 biți, sau un număr real în dublă precizie.

Generarea unor date de tip qword se face cu ajutorul directivei **QWORD**:

```
quad1 QWORD 1234567812345678h
```

Generarea unor date de tip qword se face și cu directiva **DQ** (*Define Quad – word*):

```
quad1 DQ 1234567812345678h
```

Reprezentarea în memorie a celor 8 octeți ai unui qword se face astfel încât octetul cel mai puțin semnificativ este memorat la adresa cea mai mică.

e) Ten Bytes

Valorile Ten – byte (tbyte) ocupă 10 octeți consecutivi de memorie, sau unul din registrele coprocesorului matematic.

Informația stocată într-un tbyte poate fi: un număr întreg reprezentat ca o secvență de cifre BCD (format împachetat) cu sau fără semn, sau un număr real în precizie extinsă.

Generarea unor date de tip tbyte se face cu directiva TBYTE, de ex. valoarea zecimală -1234:

```
intVal TBYTE 80000000000000001234h
```

Generarea datelor de tip tbyte se face și cu directiva DT (*Define Ten Bytes*):

```
intVal DT 80000000000000001234h
```

În format BCD împachetat fiecare cifra zecimală se reprezintă pe o tetradă (4 biți), deci 2 cifre BCD pe octet. Un întreg BCD se poate reprezenta cu maxim 19 cifre zecimale, care ocupă 76 biți. Ultima tetradă aflată la adresa cea mai mare este destinată memorării semnului.

f) Definirea datelor în virgulă mobilă

Definirea datelor în virgulă mobilă se face cu directivele:

- REAL4 – definește variabile în virgulă mobilă, în simpla precizie pe 32 biți;
- REAL8 – definește variabile în virgulă mobilă, în dubla precizie pe 64 biți;
- REAL10 – definește variabile în virgulă mobilă, cu precizie extinsă pe 80 biți.

```
rVal1 REAL4 -1.2
rVal2 REAL8 3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

Definirea datelor în virgulă mobilă se face și cu directivele:

```
rVal1 DD -1.2
rVal2 DQ 3.2E-260
rVal3 DT 4.6E+4096
```

Gama valorilor definite pot fi:

```
REAL4 - 1.18 x10-38 până 3.40 x1038
REAL8 - 2.23x10-308 până 1.79x10308
REAL10 - 3.37x10-4932 până 1.18x104932
```

2.7 Setul de instrucțiuni MASM

În cadrul acestui subcapitol, sunt prezentate în detaliu instrucțiunile de bază ale familiei de microprocesoare Intel (x86). Acolo unde este cazul, se specifică tipurile interzise de adresare.

Setul de instrucțiuni este grupat în 6 clase:

- **instrucțiuni de transfer**, care deplasează date între memorie sau porturi de intrare/ieșire și registrele microprocesorului, fără a executa nici un fel de prelucrare a datelor;
- **instrucțiuni aritmetice și logice**, care prelucrează date în format numeric;
- **instrucțiuni pentru șiruri**, specifice operațiilor cu date alfanumerice;
- **instrucțiuni pentru controlul programului**, care în esență se reduc la salturi și la apeluri de proceduri;
- **instrucțiuni specifice întreruperilor hard și soft**;

– **instrucțiuni pentru controlul procesorului.**

Această împărțire este realizată după criterii funcționale. De exemplu, instrucțiunile PUSH și POP sunt considerate ca instrucțiuni de transfer, deși, la prima vedere, ar putea fi considerate instrucțiuni specifice procedurilor. Același lucru despre instrucțiunile IN și OUT, care interfațează microprocesorul cu lumea exterioară: ele sunt considerate instrucțiuni de transfer, deși ar putea fi considerate instrucțiuni de intrare/ieșire. Intrările și ieșirile sunt însă cazuri particulare de transfer.

Fiecare categorie de instrucțiuni este însoțită de specificarea explicită a flag-urilor (indicatorilor de condiție) care sunt modificați în urma execuției.

Structura generală a instrucțiunilor x86 este următoarea:

[eticheta:] mnemonic [operanzi] ; comentariu]

Instrucțiunile pot conține zero, unu, doi sau trei operanzi. Omitem câmpurile etichetă și comentariu:

```
mnemonic
mnemonic [destinație]
mnemonic [destinație],[sursa]
mnemonic [destinație],[ sursa-1],[ sursa-2]
```

Sunt trei tipuri de bază de operanzi:

- valoare imediată
- registru (valoare se află în registru al microprocesorului)
- referință la o locație de memorie.

2.7.1 Instrucțiuni pentru transferuri de date, instrucțiuni în aritmetica binară și în aritmetica BCD. Noțiuni teoretice

Instrucțiunile de transfer permit copierea unui octet sau cuvânt de la sursa la destinație. Destinația poate fi un registru, locație de memorie sau un port de ieșire, iar sursa poate fi un registru, o locație de memorie, constante sau port de intrare. Ca regula generală destinația și sursa nu pot fi ambele locații de memorie. În specificarea sursei și destinației se vor folosi notațiile:

- segment: offset pentru adrese fizice;
- [x] paranteze partrate pentru a desemna “conținutul lui x”.

Instrucțiuni de transfer

a) Instrucțiunea MOV (Move Data).

Forma generală a instrucțiunii Mov este:

```
mov dest, sursa ; [dest] ← [sursa]
```

realizează transferul informației de la adresa efectivă data de *sursa* la *dest*.

Restricții:

- Este necesar ca ambii operanzi să fie de aceeași mărime;
- Ambii operanzi nu pot fi locații de memorie (este necesară utilizarea unui registru);
- Registrele IP, EIP, sau RIP nu pot fi ca operanzi destinație.

Structura instrucțiunii **MOV** poate fi următoarea:

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm
```

Exemple:

```
.data
var1 WORD ?
var2 WORD ?
.code
mov ax,var1
mov var2,ax
```

```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h
.code
mov eax,0 ; EAX = 00000000h
mov al,oneByte ; EAX = 00000078h
mov ax,oneWord ; EAX = 00001234h
mov eax,oneDword ; EAX = 12345678h
mov ax,0 ; EAX = 12340000h
```

```
.data
alfa dw 1234h
beta db 56h
.code
mov ax, alfa;      transfera conținutul adresei alfa în ax
mov bx, offset beta; transfera adresa efectivă alfa în bx
mov al, 75h;       transfera 75h în al
mov cx, [100];     transfera conținutul adresei 100 în cx
mov [di], bx;      transfera conținutul lui bx la adresa conținută în di
mov byte ptr alfa, [bx]; pune conținutul octetului de la adresa
                        ;dată de bx la adresa alfa
```

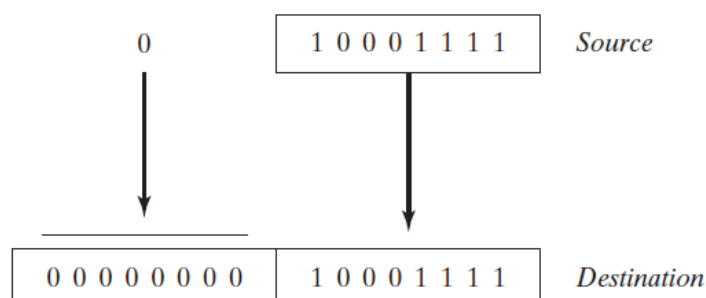
Instrucțiunea **MOVZX** (*move with zero-extend*)

Copie conținutul sursei în destinație cu extinderea valorii introducând zerouri. Această instrucțiune este utilizată numai pentru valori fără semn. Sunt trei variante:

```
MOVZX reg32,reg/mem8
MOVZX reg32,reg/mem16
MOVZX reg16,reg/mem8
```

Exemple:

```
.data
byteVal BYTE 10001111b
.code
movzx ax,byteVal ; AX = 0000000010001111b
```




```

.data
byte1 BYTE 9Bh
word1 WORD 0A69Bh
.code
movzx eax,word1 ; EAX = 0000A69Bh
movzx edx,byte1 ; EDX = 0000009Bh
movzx cx,byte1 ; CX = 009Bh

```

Instrucțiunea **MOVSX** (*move with sign-extend*)

Copie conținutul sursei în destinație cu extinderea valorii introducând unități. Această instrucțiune este utilizată numai pentru valori cu semn. Sunt trei variante:

```

MOVZX reg32,reg/mem8
MOVZX reg32,reg/mem16
MOVZX reg16,reg/mem8

```

Exemplu:

```

mov bx,0A69Bh
movsx eax,bx ; EAX = FFFFA69Bh
movsx edx,bl ; EDX = FFFFFFF9Bh
movsx cx,bl ; CX = FF9Bh

```

Exemplu de program:

```

.data
val1 WORD 1000h
val2 WORD 2000h
arrayB BYTE 10h,20h,30h,40h,50h
arrayW WORD 100h,200h,300h
arrayD DWORD 10000h,20000h
.code
main PROC
; Demonstrating MOVZX instruction:
mov bx,0A69Bh
movzx eax,bx ; EAX = 0000A69Bh
movzx edx,bl ; EDX = 0000009Bh
movzx cx,bl ; CX = 009Bh
; Demonstrating MOVSX instruction:
mov bx,0A69Bh
movsx eax,bx ; EAX = FFFFA69Bh
movsx edx,bl ; EDX = FFFFFFF9Bh
mov bl,7Bh
movsx cx,bl ; CX = 007Bh
; Memory-to-memory exchange:
mov ax,val1 ; AX = 1000h
xchg ax,val2 ; AX=2000h, val2=1000h
mov val1,ax ; val1 = 2000h
; Direct-Offset Addressing (byte array):
mov al,arrayB ; AL = 10h
mov al,[arrayB+1] ; AL = 20h
mov al,[arrayB+2] ; AL = 30h
; Direct-Offset Addressing (word array):
mov ax,arrayW ; AX = 100h

```

```

mov ax,[arrayW+2] ; AX = 200h
; Direct-Offset Addressing (doubleword array):
mov eax,arrayD ; EAX = 10000h
mov eax,[arrayD+4] ; EAX = 20000h
mov eax,[arrayD+4] ; EAX = 20000h

```

b) Instrucțiunea XCHG (Exchange Data)

Interschimbă sursa cu destinația. Forma generală:

XCHG dest, sursa

Restricții:

- registrele de segment nu pot apărea ca operanzi;
- cel puțin un operand trebuie să fie un registru general.

Sunt trei variante:

```

XCHG reg,reg
XCHG reg,mem
XCHG mem,reg

```

Exemple:

```

xchg al, ah
xchg alfa, ax
xchg sir [si], bx
xchg eax,ebx ; exchange 32-bit regs

```

Interschimbarea conținutului a doi operanzi din memorie op1 și op2 se poate face prin secvența de instrucțiuni:

```

mov reg, op1
xchg reg, op2
mov op2, reg

```

c) Instrucțiunea XLAT (Translate)

Forma generală:

XLAT

Instrucțiunea nu are operanzi, semnificația fiind:

$[al] \leftarrow ds: [[bx] + [al]]$

adică se transferă în *al* conținutul octetului de la adresa efectivă : $[bx] + [al]$.

Instrucțiunea se folosește la conversia unor tipuri de date, folosind tabele de conversie, adresa acestor tabele se introduce în *bx*, iar în *al* se introduce poziția elementului din tabel. De exemplu: conversia unei valori numerice cuprinsă între 0 și 15 în cifrele hexazecimale corespunzătoare, se poate face prin:

```
tabel BYTE '0123456789abcdef'
```

.....

```

lea bx, tabel
mov al, 11
xlat

```

În *al* se va depune cifra hexazecimală b.

c) Instrucțiunile IN (Input Data) și OUT (Output Data)

Instrucțiunea **IN** execută o citire de 8, 16, 32 biți de la portul de intrare. Și invers, instrucțiunea **OUT** execută o scriere de 8, 16, 32 biți într-un port. Sintaxa este următoarea:

```

IN accumulator, port
OUT port, accumulator

```

Port poate fi o constantă cu plaja 0 - FFh, sau poate fi o valoare încărcată în registrul DX cu plaja 0 - FFFFh. Ca **Accumulator** va fi registrul AL pentru transferuri pe 8-biți, AX pentru transferuri pe 16-biți și EAX pentru transferuri pe 32-biți.

Exemple:

```
in al,3Ch          ; input byte from port 3Ch
out 3Ch,al         ; output byte to port 3Ch
mov dx, portNumber ; DX can contain a port number
in ax,dx           ; input word from port named in DX
out dx,ax          ; output word to the same port
in eax,dx          ; input doubleword from port
out dx,eax         ; output doubleword to same port
```

f) Instrucțiunea LEA (Load Effective Address)

Are ca efect încărcarea adresei efective (offsetul) într-un registru general.

Forma generală:

LEA reg, sursa

unde:

- *sursa* - este un operand aflat în memorie, specificat printr-un mod de adresare ;
- *reg* - este un registru general.

Exemplu:

```
lea bx, alfa
lea si, alfa [bx][si]
```

Același efect se obține folosind operandul OFFSET în Instrucțiunea MOV:

```
mov bx, offset alfa
mov si, offset alfa [bx][si]
```

g) Instrucțiunea LDS/ LES (Load Data Segment/ Load Extra Segment)

Forma generală:

LDS reg, sursa

unde:

- *reg* - este un registru general de 16 biți;
- *sursa* - este un operand de tip double – word aflat în memorie, care conține o adresă completă de 32 biți.

Are ca efect transferul unei adrese complete în perechea de registre *ds* și *reg* specificat în instrucțiune, adică:

```
[reg] ← [[sursa]]
[ds]  ← [[sursa] + 2]
```

Exemplu:

```
alfa byte 25
adr_alfa dword alfa
```

.....

```
lds si, adr_alfa ; în registru si se transferă offset-ul,
                  ; iar în ds adresa de segment a celulei alfa
mov byte ptr [si], 75
```

.....

i) Instrucțiunea LAHF (Load AH with FLAGS)

Instrucțiunea încarcă în registrul AH octetul dat de partea cea mai puțin semnificativă a registrului FLAGS, ce conține indicatorii. Instrucțiunea nu are operanzi.

$AH \leftarrow \text{FLAGS}_{0..7}$

Ex.

```
.data
saveflags BYTE ?
.code
lahf ; load flags into AH
mov saveflags,ah ; save them in a variable
```

j) Instrucțiunea SAHF (Store AH into FLAGS)

Instrucțiunea încarcă în registrul FLAGS (EFLAGS or RFLAGS), în octetul cel mai puțin semnificativ conținutul registrului AH, adică:

$$\text{FLAGS} \leftarrow [\text{AH}]_{0.7}$$

Instrucțiunea nu are operanzi.

Ex.

```
mov ah,saveflags    ; load saved flags into AH
sahf                ; copy into Flags register
```

Instrucțiunea PUSH

Instrucțiunea PUSH decrementează registrul ESP și copie operandul sursă în stivă. Un operand pe 16 biți decrementează registrul ESP cu 2, iar un operand pe 32 biți – cu 4. Sunt 3 formate ale instrucțiunii:

```
PUSH reg/mem16
PUSH reg/mem32
PUSH imm32
```

Instrucțiunea POP

Instrucțiunea POP copie conținutul stivei în operandul sursă pe 16 sau 32 biți și incrementează registrul ESP cu valorile 2 sau 4 respectiv. Sunt 2 formate ale instrucțiunii:

```
POP reg/mem16
POP reg/mem32
```

Instrucțiunile PUSHFD și POPFD

Instrucțiunea PUSHFD copie conținutul registrului EFLAGS pe 32 biți în stivă, iar POPFD extrage din stivă valoarea pe 32 biți și încarcă registrul de fanioane EFLAGS.

```
pushfd
popfd
```

Instrucțiunile PUSHAD, PUSHA, POPAD și POPA

Instrucțiunea PUSHAD copie conținutul registrelor de uz general în stivă în ordinea următoare:

EAX, ECX, EDX, EBX, ESP (valorile înaintea execuției PUSHAD), EBP, ESI și EDI. Iar instrucțiunea POPAD extrage din stivă și încarcă regiștrii de uz general în ordinea inversă.

Respectiv instrucțiunea PUSHA copie conținutul registrelor de uz general pe 16 biți în stivă în ordinea următoare: AX, CX, DX, BX, SP, BP, SI, DI. Iar instrucțiunea POPA extrage din stivă și încarcă regiștrii de uz general pe 16 biți, în ordinea inversă.

2.7.2 Aritmetica binara

Aceste instrucțiuni modifica conținutul registrului FLAGS.

a) Instrucțiunea ADD (Add)

Forma generală:

$$\text{ADD dest, sursa ; } [\text{dest}] \leftarrow [\text{dest}] + [\text{sursa}]$$

unde:

- *dest* poate fi un registru general sau o locație de memorie;
- *sursa* poate fi un registru general, o locație de memorie sau o constantă.

Operanzii au aceeași structură ca la instrucțiunea MOV. Cei doi operanzi nu pot fi simultan locații de memorie.

Operația se poate efectua pe 8,16, 32 sau pe 64 biți. Cei doi operanzi trebuie să aibă aceeași dimensiune (același tip). În caz de ambiguitate se va folosi operatorul PTR.

Indicatorii afectați sunt: AF, CF, PF, SF, ZF și OF

Exemple:

```

add    ax, 5
add    bl, 5
add    ax, bx
add    word ptr [bx], 75
add    alfa, ax
add    alfa, 5
add    byte ptr [si], 75
add    byte ptr alfa, 75

```

```

.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
mov eax,var1 ; EAX = 10000h
add eax,var2 ; EAX = 30000h

```

```

.data
sum qword 0

```

```

.code
    mov    rax,5
    add    rax,6
    mov    sum,rax

```

b) Instrucțiunea ADC (Add with Carry)

Forma generală:

ADC *dest*, *sursa* ; $[dest] \leftarrow [dest] + [sursa] + [CF]$

Unde *dest* și *sursa* au aceeași semnificație ca la instrucțiunea ADD, iar CF este Carry Flag.

Instrucțiunea adună conținutul *dest* cu conținutul *sursei* și cu bitul de transport CF. Indicatorii afectați sunt aceiași de la instrucțiunea ADD.

Operația ADC se folosește la adunări de operanți pe mai multe cuvinte, operație în care poate apărea transport de care trebuie să se țină seama.

Exemplu 1. Să se adune două numere op1, op2 pe 2 cuvinte (dword).

```

op1    dword 12345678h
op2    dword 0abcdefgh
rez    dword ?

```

```

.....
mov    ax, word ptr op1
add    ax, word ptr op2
mov    word ptr rez, ax
mov    ax, word ptr op1+2
adc    ax, word ptr op2+2; se considera eventualul transport
mov    word ptr rez+2, ax

```

Exemplu 2. Adunăm 2 numere întregi pe 8 biți (FFh + FFh), producând un rezultat pe 16-biți, suma va fi în DL:AL, care este 01FEh:

```

mov    dl,0
mov    al,0FFh
add    al,0FFh ; AL = FEh
adc    dl,0 ; DL/AL = 01FEh

```

c) Instrucțiunea SUB (Substrat)

Forma generală:

SUB dest, sursa ; $[dest] \leftarrow [dest] - [sursa]$

unde *dest* și *sursa* au aceeași semnificație ca la instrucțiunea ADD. Indicatorii afectați sunt cei specificați la ADD. Structura operanzilor ca la instrucțiunea MOV.

```
.data
var1 DWORD 30000h
var2 DWORD 10000h
.code
mov eax,var1 ; EAX = 30000h
sub eax,var2 ; EAX = 20000h
```

d) Instrucțiunea SBB (Substrat with Borrow)

Forma generală:

SBB dest, sursa ; $[dest] \leftarrow [dest] - [sursa] - [CF]$

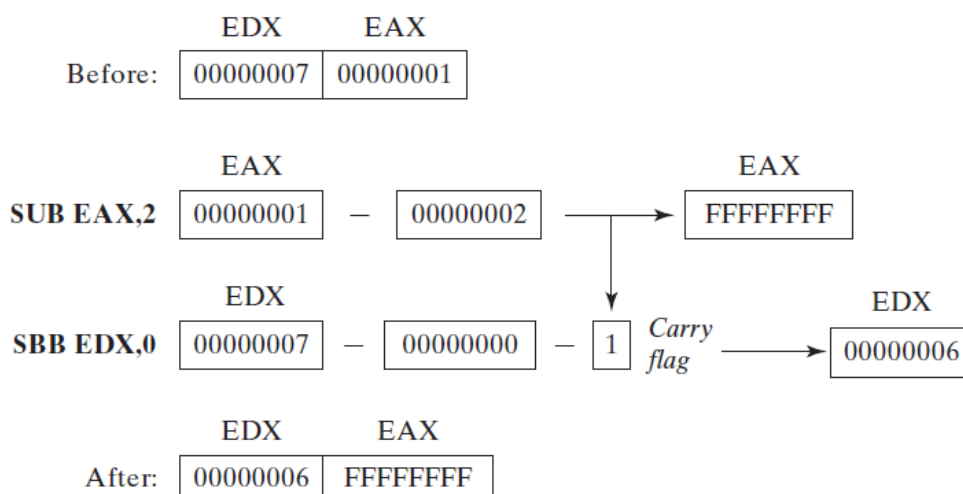
unde semnificația *dest*, *sursa* și CF sunt cele prezentate la ADC. Instrucțiunea SBB ia în considerare eventualul împrumut. Exemplu:

```
Op1  dword 12345678h
Op2  dword 0abcdef78h
Rez  dword ?

.....
mov  ax, word ptr op1
sub  ax, word ptr op2
mov  word ptr rez, ax
mov  ax, word ptr op1 + 2
sbb  ax, word ptr op2 + 2 ; se considera eventualul împrumut
mov  word ptr rez + 2, ax
```

Alt exemplu. Scăderea dintr-un întreg de 64 biți un untreg de 32 biți. Perechea de regiștri EDX:EAX se încarcă cu valoarea 0000000700000001h și scădem 2. La prima etapă se va scădea din partea inferioară (EAX-00000001h), adică din 1 vom scădea 2, ce va duce la împrumut cu setarea indicatorului Carry.

```
mov edx,7 ; partea superioară
mov eax,1 ; partea inferioară
sub eax,2 ; scăderea 2
sbb edx,0 ; scăderea părții superioare
```

**e) Instrucțiunea INC (Increment)**

Forma generală:

INC dest ; $[dest] \leftarrow [dest] + 1$

unde *dest* este un registru general, un operand din memorie. Semnificația fiind operandul *dest* este incrementat cu unu. Indicatorii afectați sunt AF, PF, SF, ZF, OF.

Exemple:

```

inc    alfa
inc    bl
inc    eax
inc    rbx
inc    word ptr [bx] [si]

```

f) Instrucțiunea DEC (decrement)

Forma generală:

DEC *dest* ; $[dest] \leftarrow [dest] - 1$

unde *dest* are aceeași semnificație ca *dest* de la Instrucțiunea INC. Aceeași indicatori ca la INC sunt afectați.

g) Instrucțiunea NEG (Negate)

Forma generală:

NEG *dest* : $[dest] \leftarrow 0 - [dest]$ schimbare de semn

unde *dest* este un operand pe 8, 16, 32 sau 64 biți ce poate fi un registru general sau o locație de memorie. Instrucțiunea afectează indicatorii AF, CF, PF, SF, OF și ZF.

Exemplu:

```

alfa    byte    75
.....
mov     al, alfa
neg     al
mov     alfa, al ; la adresa alfa avem - 75
sau
neg     alfa

```

h) Instrucțiunea CMP (Compare)

Forma generală:

CMP *dest*, *sursa* ; $[dest] - [sursa]$

Instrucțiunea realizează o operație de scădere între cei doi operanzi, fără a modifica operandul *dest* sau *sursa* cu poziționarea indicatorilor de condiție. Indicatorii afectați sunt: AF, CF, PF, SF, ZF și OF.

Aceasta instrucțiune se folosește împreună cu instrucțiunea de salt condiționat.

Când comparăm doi operanzi fără semn indicatoarele Zero și Carry indică următoarea relație între operanzi:

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

Exemple:

```

cmp     ax, alfa[bx][si]
cmp     [si], 0

```

i) Instrucțiunea CBW (Convert Byte to Word)

Are ca efect extinderea bitului de semn (AL_7) din AL la întreg registru AH, adică:

dacă bitul de semn $AL_7 = 0$ atunci $[ah] \leftarrow 00h$
 altfel $[ah] \leftarrow 0ffh$.

`MUL reg/mem32`

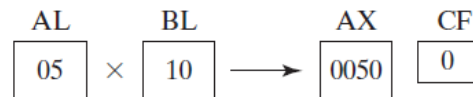
unde *reg* poate fi un registru sau o locație de memorie *mem* de 8, 16, 32 biți. Rezultatul se obține pe un număr dublu de biți (16, 32, 64). Operația realizată este produsul între acumulator și sursa cu depunerea rezultatului în acumulatorul extins. Cei doi operanzi se consideră numere fără semn.

Dacă *sursa* este pe octet avem:

$[AX] \leftarrow [AL] * [reg/mem8]$

```
mov al,5h
mov bl,10h
mul bl ; AX = 0050h, CF = 0
```

Diagrama ilustrează interacțiunea dintre registre:



Dacă *sursa* este pe cuvânt avem:

$[DX:AX] \leftarrow [AX] * [reg/mem16]$

```
.data
val1 WORD 2000h
val2 WORD 0100h
.code
mov ax,val1 ; AX = 2000h
mul val2 ; DX:AX = 00200000h, CF = 1
```

Diagrama ilustrează interacțiunea dintre registre:



Dacă *sursa* este pe 32 biți avem:

$[EDX:EAX] \leftarrow [EAX] * [reg/mem32]$

```
mov eax,12345h
mov ebx,1000h
mul ebx ; EDX:EAX = 0000000012345000h, CF = 0
```

Diagrama ilustrează interacțiunea dintre registre:



iar dacă *sursa* este pe 64 biți avem:

$[RDX:RAX] \leftarrow [RAX] * [reg/mem64]$

```
mov rax,0FFFF0000FFFF0000h
mov rbx,2
mul rbx ; RDX:RAX = 00000000000000001FFFE0001FFFE0000
```

Afectează indicatorii CF și OF, ceilalți sunt nedefiniți.

I) Instrucțiunea IMUL (Integer Multiply)

Instrucțiunea IMUL semnifică înmulțirea cu semn. Instrucțiunea poate avea 1, 2, sau 3 operanzi.

Afectează indicatorii CF și OF, restul sunt nedefiniți.

Structura cu un operand:

```
IMUL reg/mem8 ; AX = AL * reg/mem8
IMUL reg/mem16 ; DX:AX = AX * reg/mem16
IMUL reg/mem32 ; EDX:EAX = EAX * reg/mem32
```

Structura cu doi operanzi. Structura cu doi operanzi **trunchiază** produsul la lățimea registrului de destinație. Dacă cifrele semnificative sunt pierdute, se setează indicatorii CF și OF.

```
IMUL reg16, reg/mem16
IMUL reg16, imm8
IMUL reg16, imm16

IMUL reg32, reg/mem32
IMUL reg32, imm8
IMUL reg32, imm32
```

Structura cu 3 operanzi - $op1 = op2 * op3$ (**trunchiază** produsul):

```
IMUL reg16, reg/mem16, imm8
IMUL reg16, reg/mem16, imm16

IMUL reg32, reg/mem32, imm8
IMUL reg32, reg/mem32, imm32
```

Exemplu cu 2 operanzi:

```
.data
word1 SWORD 4
dword1 SDWORD 4
.code
mov ax, -16      ; AX = -16
mov bx, 2        ; BX = 2
imul bx, ax      ; BX = -32
imul bx, 2       ; BX = -64
imul bx, word1   ; BX = -256
mov eax, -16     ; EAX = -16
mov ebx, 2       ; EBX = 2
imul ebx, eax    ; EBX = -32
imul ebx, 2      ; EBX = -64
imul ebx, dword1 ; EBX = -256
```

Exemplu cu 3 operanzi:

```
.data
word1 SWORD 4
dword1 SDWORD 4
.code
imul bx, word1, -16      ; BX = word1 * -16
imul ebx, dword1, -16    ; EBX = dword1 * -16
imul ebx, dword1, -2000000000 ; signed overflow!
```

m) Instrucțiunea DIV (Divide)

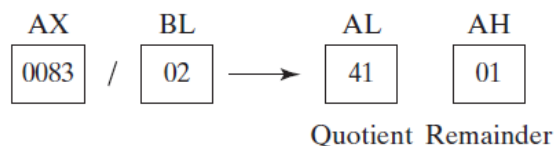
Forma generală:

DIV sursa

unde *sursa* este un registru sau o locație de memorie, reprezentată pe octet, cuvânt, 32 biți sau 64 biți. Instrucțiunea realizează împărțirea fără semn între deîmpărțit și împărțitor. Dacă împărțitorul (*sursa*) este reprezentat pe octet atunci deîmpărțitul este AX și rezultatul este: câtul în **al** iar restul în **ah**, adică:

DIV sursa ; [al] \leftarrow [ax]/ [sursa]
 ; [ah] \leftarrow restul împărțirii [ax]/ [sursa]

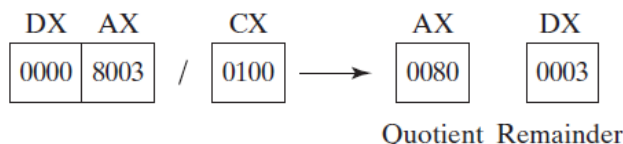
```
mov ax,0083h ; deîmpărțitul
mov bl,2 ; împărțitorul
div bl ; AL = 41h-catul, AH = 01h-restul
```



Dacă împărțitorul (sursa) este reprezentat pe cuvânt atunci deîmpărțitul este considerat în DX și AX, câtul se obține în AX iar restul în DX, adică

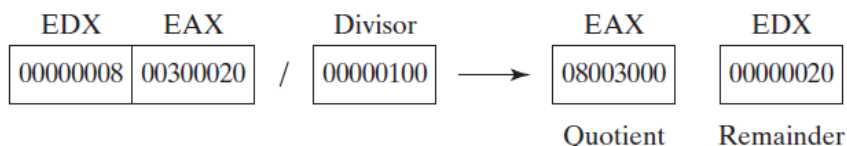
DIV sursa ; [ax] \leftarrow câtul împărțirii [dx:ax]/[sursa]
 ; [dx] \leftarrow restul împărțirii [dx:ax]/[sursa]

```
mov dx,0 ; clear deîmpartitul, high
mov ax,8003h ; deîmpartitul, low
mov cx,100h ; împartitorul
div cx ; AX = 0080h-câtul, DX = 0003h-restul
```



Dacă împărțitorul (sursa) este reprezentat pe 32 biți atunci deîmpărțitul este considerat în EDX și EAX (64 biți), câtul se obține în EAX iar restul în EDX

```
.data
dividend QWORD 0000000800300020h
divisor DWORD 00000100h
.code
mov edx,DWORD PTR dividend + 4 ; high doubleword
mov eax,DWORD PTR dividend ; low doubleword
div divisor ; EAX = 08003000h, EDX = 00000020h
```



Dacă împărțitorul (sursa) este reprezentat pe 64 biți atunci deîmpărțitul este considerat în RDX și RAX (64 biți), câtul se obține în RAX iar restul în RDX

```
.data
dividend_hi QWORD 00000000000000108h
dividend_lo QWORD 0000000033300020h
divisor QWORD 0000000000010000h
.code
mov rdx,dividend_hi
mov rax,dividend_lo
div divisor ; RAX = 0108000000003330
; RDX = 00000000000000020
```

Toți indicatorii nu sunt definiți. Operația de împărțire poate conduce la depășiri, dacă câtul depășește valoarea maximă reprezentabilă pe 8, respectiv pe 16 biți sau dacă împărțitorul este 0.

n) Instrucțiunea IDIV (Integer Divide)

Forma generală:

IDIV sursa

Semnificația instrucțiunii și a operandului *sursa* este aceeași ca la Instrucțiunea DIV, cu o singură diferență importantă – deîmpărțitul cu semn trebuie să fie extins înainte ca împărțirea să fie executată.

Extinderea semnului se execută cu instrucțiunile **CBW**, **CWD**, **CDQ**.

Indicatorii sunt nedefiniți. Operația poate conduce la depășiri.

Ex. Împărțim -48 la +5

```
.data
byteVal SBYTE -48 ; D0 hexadecimal
.code
mov al,byteVal ; partea inferioară a deîmpărțitului
cbw           ; extindem AL în AH
mov bl,+5     ; împărțitorul
idiv bl       ; AL = -9 - câtul, AH = -3 - restul
```

2.7.3 Aritmetica BCD

2.3.1 Instrucțiunea AAA (ASCII Adjust for Addition)

Instrucțiunea nu are operanzi și execută corecția acumulatorului AX, după operații de adunare cu numere în format BCD despachetat. Semnificația este:

```
daca [AL0:3] > 9 sau [AF] = 1, atunci {
    [AL] ← [AL] + 6
    [AH] ← [AH]+1
    [AF] ← 1
    [CF] ← 1
    [AL] ← [AL] AND 0FH
}
```

Indicatorii afectați : AF, CF, restul nedefiniți.

Exemplu:

```
mov ax, 408h
mov dx, 209h
add ax, dx ; [ax]=0611h
AAA       ; [ax]=0707h
```

2.3.2 Instrucțiunea AAS (ASCII Adjust for Subtraction)

Instrucțiunea nu are operanzi și execută corecția acumulatorului AX, după operații de scădere cu numere în format BCD despachetat. Semnificația este:

```
daca [AL0:3] > 9 sau [AF] = 1, atunci {
    [AL] ← [AL] - 6
    [AH] ← [AH] - 1
    [AF] ← 1
    [CF] ← 1
    [AL] ← [AL] AND 0FH
}
```

Indicatorii afectați : AF, CF, restul nedefiniți.

Exemplu:

```
mov ax, 408h
mov dx, 209h
sub ax, dx ; [ax]=01ffh
AAS       ; [ax]=0109h
```

2.3.3 Instrucțiunea DAS (Decimal Adjust for Subtraction)

Instrucțiunea nu are operanzi și execută corecția zecimala a acumulatorului AL, după operații de scădere cu numere în format BCD împachetat. Semnificația este:

```
daca [AL0:3] > 9 sau [AF] = 1, atunci {
    [AL] ← [AL] - 6
    [AF] ← 1
}
```

```

    }
    daca acum [AL4:7] > 9 sau CF = 1, atunci {
        [AL] ← [AL] - 60H
        [CF] ← 1
    }

```

Indicatorii afectați : AF, CF, PF, SF, ZF. Indicatorul OF este nedefinit.

De exemplu, în urma secvenței:

```

MOV  AL, 52H
SUB  AL, 24H    ; AL = 2EH
DAS          ; AL = 28H

```

se obține în AL rezultatul corect 28H.

2.3.5 Instrucțiunea AAM (ASCII Adjunct for Multiply)

Instrucțiunea nu are operanzi și efectuează o corecție a acumulatorului AX, după o înmulțire pe 8 biți cu operanzi în format BCD despachetat.

Semnificația este următoarea:

```

[AH] ← [AL] / 10
[AL] ← [AL] MOD 10

```

Indicatori afectați: PF, SF, ZF, restul nedefinite.

De exemplu

```

mov al, 7
mov bl, 9
mul bl ; 003Fh
AAM   ; 0603h

```

2.3.6 Instrucțiunea AAD (ASCII Adjunct for Division)

Instrucțiunea nu are operanzi și efectuează o corecție a acumulatorului AX, înaintea unei împărțiri a doi operanzi în format BCD despachetat.

Semnificația este următoarea:

```

[AL] ← [AH] * 10 + [AL]
[AH] ← 0

```

Indicatori afectați: PF, SF, ZF, restul nedefinite.

De exemplu

```

mov ax, 305h
mov bl, 2
AAD   ; [ax]=35h
div bl ; [al]=12h [ah]=1

```

2.7.4 Exemple programe

Exemplul 1. Acest exemplu prezintă câteva tehnici de adresare specifice procesoarelor din familia x86:

```

INCLUDE Irvine32.inc
.data
alfa WORD 3 DUP(?)
.code
main proc
mov  ax,17 ; Adresare imediata a operandului
          ; sursa care este o constantă zecimala
mov  ax,10101b ; Sursa este o constantă binara
mov  ax,11b    ;
mov  ax,0bch   ; Sursa este o constantă hexa

```

```

mov  alfa,ax      ; Adresare directa a operandului destinatie
mov  cx,ax        ; Interschimba registrele ax si bx
mov  ax,bx        ; Folosind registrul cx
mov  ax,cx        ;
xchg ax,bx        ; Interschimba direct cele 2 registre.
mov  si,2
mov  alfa[si],ax  ; Adresare relativa cu registrul
                        ; și a operandului destinație
mov  esi,2
mov  ebx,offset alfa ; Adresare imediată a operandului
                        ; sursă (adresa variabilei alfa)
                        ; datorită operatorului OFFSET
lea  ebx,alfa     ; Acelasi efect
mov  ecx,[ebx][esi] ; Adresare bazata indexata a sursei
mov  cx,alfa[2]    ; Acelasi efect.
mov  cx,[alfa+2]   ; Acelasi efect
mov  di,4
mov  byte ptr [ebx][edi],55h ; Se va folosi această
                        ; variantă când se dorește o
                        ; adresare la nivel de octet

mov  esi,2
mov  ebx,3
mov  alfa[ebx][esi],33h ; Adresare bazata indexata
                        ; relativa a operandului destinație
mov  alfa[ebx+esi],33h ; Notatii echivalente
mov  [alfa+ebx+esi],33h
mov  [ebx][esi]+alfa,33h

exit
main ENDP
END main

```

Pentru exemplificări s-a folosit Instrucțiunea *mov* în diferite variante de adresare: registru la registru, din memorie în registru și din registru în memorie.

Exemplul 2. Să se calculeze expresia aritmetică: $e=((a+b*c-d)/f+g*h)/i$. Se consideră *a, d, f* – cuvânt, iar *b, c, g, h, i* –byte. Ca să putem executa împărțirea cu *f* convertim împărțitorul la dublucuvânt. Ne vor interesa doar câturile împărțirilor, rezultatul va fi de tip octet.

```
INCLUDE Irvine32.inc
```

```
.data
```

```

a dw 5
b db 6
cd db 10
d dw 5
f dw 6
g db 10
h db 11
i db 10
interm dw ?
rez db ?

```

```
.code
```

```
main proc
```

```

mov eax,0
mov al, b

```

```

    imul cd          ; in ax avem b*c
    add ax, a        ; ax=b*c+a
    sub ax, d        ; ax=b*c+a-d
    cwd             ; am convertit cuvantul din ax, in dublu cuvantul , retinut in dx:ax
    idiv f           ; obtinem catul in ax si restul in dx ax=(a+b*c-d)/f
    mov interm, ax   ; interm=(a+b*c-d)/f
    mov al, g
    imul h           ; ax=g*h
    add ax, interm    ; ax=(a+b*c-d)/f+g*h
    idiv i           ; se obtine catul in al si restul in ah
    mov rez, al

    exit
main ENDP
END main

```

Fișierul Irvine32.inc este o bibliotecă de proceduri ce apelează funcții Win32 API, concepută pentru a oferi o interfață simplă pentru intrări-ieșiri de date la consolă. (API - Application Programming Interface, un set de funcții oferite de sistemul de operare Windows pentru manipularea resurselor calculatorului și sunt implementate în următoarele trei biblioteci: user32.dll, kernel32.dll și gdi32.dll).

Fereastra consolei (sau fereastra – linie de comandă, cmd.exe) este o fereastră textuală, creată de MS-Windows, pentru afișarea liniei de comandă. Vom descrie unele funcții utilizate pentru a introduce date de la tastatură și afișarea datelor de ieșire. Aceste proceduri se apelează cu instrucțiunea *call*, de exemplu *call Crlscr*.

Procedurile apelate utilizează diferite echivalări.

2.7.5 Echivalări și operatori

Echivalările reprezintă niște valori constante de tip constantă numerică sau șir de caractere atribuite unor nume simbolice, simbolurile putând fi înlocuite în locul valorilor respective.

a) Echivalările numerice sunt folosite pentru a atribui o constantă numerică unui simbol. Acestea pot fi:

- redefinibile: valoarea unui simbol poate fi redefinită în orice moment în timpul asamblării;

Sintaxa unei echivalări numerice redefinite este:

nume = expresie

unde: - **expresie** poate fi un întreg, o expresie constantă, o constantă de tip șir de caractere sau două constante sau o expresie evaluată la o adresă, **nume** este un nume de simbol unic sau un nume de simbol definit anterior cu =.

- neredefinibile: valoarea simbolului nu poate fi redefinită în timpul asamblării.

Sintaxa unei echivalări numerice neredefinibile este:

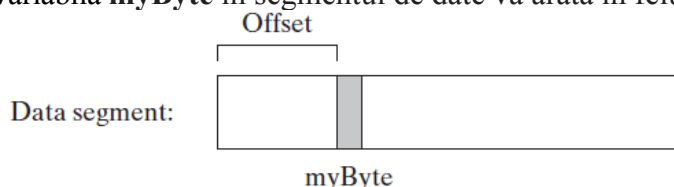
nume EQU expresie

Parametrii având semnificațiile: **nume** este un nume de simbol unic.

Simbolurile definite prin echivalări numerice pot fi folosite în construcții ulterioare ca operanzi imediați. Acestor simboluri nu li se alocă memorie.

Operatorii admiși în cazul folosirii expresiilor constante sunt operatori aritmetici: +, -, *, /(împărțire întreagă) și mod.

OFFSET. Operatorul **OFFSET** returnează distanța unei variabile în octeți, de la începutul segmentului. De exemplu variabila **myByte** în segmentul de date va arăta în felul următor:



PTR. Operatorul **PTR** este utilizat la accesarea unui operand, dimensiunea cărui este diferită de cea necesară. Exemple de utilizare.

```

.data
myDouble DWORD 12345678h
.code
mov ax,WORD PTR myDouble ; 5678h
mov ax,WORD PTR [myDouble+2] ; 1234h
mov bl,BYTE PTR myDouble ; 78h

.data
wordList WORD 5678h,1234h
.code
mov eax,DWORD PTR wordList ; EAX = 12345678h

```

TYPE. Operatorul **TYPE** întoarce un număr ce reprezintă tipul unei expresii în octeți.

LENGTHOF. Operatorul **LENGTHOF** întoarce numărul de elemente a unui șir, tablou de date.

SIZEOF. Operatorul **SIZEOF** întoarce numărul total de octeți alocați pentru un tablou sau variabilă definită cu DUP.

Exemple.

```

.DATA
intgr = 14*3 ;=42
intgr = intgr/4 ;10
intgr = intgr mod 4 ;2
intgr = intgr+4 ;6
intgr = intgr-3 ;3
m1 EQU 5
m2 EQU -5
const EQU m1+m2 ;const=0
vect DW 60 DUP(?)
s_vect EQU SIZEOF vect ;60 * 2 = 120
l_vect EQU LENGTHOF vect ;60
t_vect EQU TYPE vect ;2
verif EQU t_vect*l_vect ;=2*60
mov ax, SIZEOF vect

```

În general echivalările sunt păstrate în fișiere separate de includere, fiind incluse într-un program prin intermediul directivei INCLUDE.

2.7.6 Procedurile utilizate

Procedura	Descrierea
Clrscr	Șterge fereastra consolei și poziționează cursorul în colțul stânga sus.
Crlf	Sunt coduri ASCII ce indică sfârșitul liniei, din rând nou
Delay	Întreține execuția programului pentru un interval specificat de milisecunde
DumpMem	Procedura afișează un tablou de date (array) în hexazecimal
DumpRegs	Afișează conținutul regiștrilor EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, și registrul EIP în hexazecimal
Gotoxy	Plasează cursorul pe un rând și coloană în fereastra de consolă
Random32	Procedura generează și returnează un număr întreg aleator pe 32 de biți în EAX.
RandomRange	Generează un întreg aleator într-un interval specificat.
ReadChar	Așteaptă un singur caracter introdus de la tastatură și returnează caracterul în AL
ReadDec	Citește de la tastatură un întreg zecimal fără semn pe 32 de biți, finalizarea introducerii - tasta Enter
ReadHex	Citește de la tastatură un întreg hexazecimal pe 32 de biți, finalizarea introducerii - tasta Enter
ReadInt	Citește de la tastatură un întreg zecimal cu semn pe 32 de biți, finalizarea

	introducerii - tasta Enter
<i>ReadString</i>	Citește de la tastatură un șir de caractere, finalizarea introducerii - tasta Enter
<i>SetTextColor</i>	Setează culorile textului și de fundal a consolei.
<i>WaitMsg</i>	Afișează un mesaj și așteaptă un clic pe o tastă
<i>WriteBin</i>	Afișează un întreg fără semn pe 32 de biți în format binar ASCII
<i>WriteBinB</i>	Afișează un întreg în format binar pe un octet, cuvânt, sau 32 de biți
<i>WriteChar</i>	Afișează un singur caracter
<i>WriteDec</i>	Afișează un întreg fără semn pe 32 de biți în format zecimal
<i>WriteHex</i>	Afișează un întreg pe 32 de biți în format hexazecimal
<i>WriteHexB</i>	Afișează un întreg de un byte, word, sau doubleword în format hexazecimal
<i>WriteInt</i>	Afișează un întreg cu semn pe 32 de biți în format zecimal
<i>WriteString</i>	Afișează un șir, finalizat cu un octet nul
<i>WriteWindowsMsg</i>	Afișează un șir care conține cele mai recente erori generate de MS-Windows

Descrierea detaliată

Delay. Înainte de a apela *Delay*, setați registrul EAX cu intervalul dorit în milisecunde. Exemplu:

```
mov eax,1000 ; 1 second
call Delay
```

DumpMem. Înainte de apelare, în ESI - încărcăți deplasamentul tabloului, în ECX - numărul de locații de memorie, iar în EBX – tipul locației de memorie (1 = byte, 2 = word, 4 = doubleword). În exemplul următor se afișează un tablou din 11 cuvinte duble (doubleword) în hexazecimal:

```
.data
array DWORD 1,2,3,4,5,6,7,8,9,0Ah,0Bh
.code
main PROC
mov esi,OFFSET array ; starting OFFSET
mov ecx,LENGTHOF array ; numărul de locații de memorie
mov ebx,TYPE array ; tipul locației de memorie - doubleword
call DumpMem
```

La ieșire se va afișa:

```
00000001 00000002 00000003 00000004 00000005 00000006
00000007 00000008 00000009 0000000A 0000000B
```

DumpRegs. Afișează conținutul regiștrilor EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, și registrul EIP în hexazecimal. Se afișează, de asemenea, valorile fanioanelor (indicatoarelor) Carry, Sign, Zero, Overflow, Auxiliary Carry, și Parity.

```
call DumpRegs
```

La ieșire se va afișa:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0 AF=0 PF=1
```

Gotoxy. Plasează cursorul pe un rând și coloană în fereastra de consolă. În mod implicit, valorile coordonatei X este de la 0-79 și Y- de la 0 la 24. Când apelați *Gotoxy*, încărcăți coordonatele Y (rând) în DH și coordonatele X (coloana) în DL. Exemplu:

```
mov dh,10 ; randul 10
mov dl,20 ; coloana 20
call Gotoxy ; pozitia cursorului
```

Random32. Procedura *Random32* generează și returnează un întreg aleator pe 32 de biți în EAX. Când Procedura este solicitată în mod repetat, *Random32* generează o secvență aleatoare de valori.

```
.data
```

```

    randVal DWORD ?
    .code
    call Random32
    mov randVal,eax

```

RandomRange. Procedura RandomRange produce un număr întreg aleator în intervalul de la 0 la n-1, unde n este un parametru de intrare încărcat în registrul EAX. Numărul generat aleator este întors în EAX. Următorul exemplu generează un număr întreg aleator între 0 și 4999 și se salvează într-o variabilă numită *randVal*.

```

    .data
    randVal DWORD ?
    .code
    mov eax,5000
    call RandomRange
    mov randVal,eax

```

ReadChar. Procedura ReadChar așteaptă un singur caracter introdus de la tastatură și îl încarcă în registrul AL. Caracterul este introdus fără ecou (fără afișare).

```

    .data
    char BYTE ?
    .code
    call ReadChar
    mov char,al

```

Dacă executați clic pe o tastă funcțională (F1 ...), săgeată (← ...), Ins, sau Del, procedura încarcă registrul AL cu zero, și în AH se va introduce scan -codul tastei.

ReadDec. Procedura ReadDec citește un întreg zecimal de 32-biți fără semn de la tastatură și încarcă valoarea în EAX. Spațiile sunt ignorate. Se introduc numai valori zecimale. De exemplu, dacă utilizatorul introduce 123ABC, valoarea returnată în EAX este 123.

```

    .data
    intVal DWORD ?
    .code
    call ReadDec
    mov intVal,eax

```

ReadHex. Procedura ReadHex citește un întreg hexazecimal de 32 de biți de la tastatura și încarcă valoarea binară corespunzătoare în EAX. Nu indică eroare la introducerea valorilor nevalide. Puteți utiliza atât litere mari și mici pentru caracterele de la A la F. Un număr maxim de opt cifre pot fi introduse (caracterele suplimentare sunt ignorate). Spațiile sunt ignorate.

```

    .data
    hexVal DWORD ?
    .code
    call ReadHex
    mov hexVal,eax

```

ReadInt. Procedura ReadInt citește un întreg de 32-biți, cu semn, de la tastatură și încarcă valoarea în EAX. Utilizatorul poate introduce, opțional, semnul plus sau minus, și restul numărului poate consta doar din cifre. ReadInt setează în „1” flag-ul *Overflow* și va afișa un mesaj de eroare în cazul în care valoarea introdusă nu intră în plaja numerelor cu semn de 32-biți (plaja: -2,147,483,648 +2,147,483,647). Se introduc numai valori zecimale. De exemplu, în cazul în care utilizatorul introduce 123ABC, valoarea introdusă va fi 123.

```

    .data
    intVal SDWORD ?
    .code
    call ReadInt
    mov intVal,eax

```

ReadString. Procedura ReadString citește un șir de caractere de la tastatură, clic Enter – finalizarea introducerii. Înainte de invocarea procedurii, este necesar să încarcăm EDX cu offset-ul buffer-ului unde va fi stocat șirul și ECX cu numărul maxim de caractere care va fi introdus, plus 1 (pentru byte-ul nul de finalizare). Procedura returnează numărul de caractere introduse de utilizator în EAX.

```
.data
buffer BYTE 21 DUP(0) ; input buffer
byteCount DWORD ? ; holds counter
.code
mov edx,OFFSET buffer ; point to the buffer
mov ecx,SIZEOF buffer ; specify max characters
call ReadString ; input the string
mov byteCount,eax ; number of characters
```

ReadString introduce automat byte-ul nul în memorie, la sfârșitul șirului. După ce utilizatorul a introdus șirul "ABCDEFGH" în variabila *buffer* va fi:

41 42 43 44 45 46 47 00	ABCDEFGH
-------------------------	----------

Conținutul variabilei *byteCount* va fi egală cu 7.

SetTextColor. Procedura *SetTextColor* setează culorile textului și de fundal a consolei. La invocarea procedurii culorile textului și de fundal este necesar să fie încărcate în registrul EAX. Constantele culorilor predefinite sunt următoarele:

black = 0	red = 4	gray = 8	lightRed = 12
blue = 1	magenta = 5	lightBlue = 9	lightMagenta = 13
green = 2	brown = 6	lightGreen = 10	yellow = 14
cyan = 3	lightGray = 7	lightCyan = 11	white = 15

Constantele sunt predefinite în *irvine32.inc*. Următoarele constante indică culoarea galben a caracterului pe fundal albastru:

```
yellow _ (blue * 16)
```

Următoarea secvență setează: caracter alb pe fundal albastru:

```
mov eax,white _ (blue * 16) ; white on blue
call SetTextColor
```

WaitMsg. Procedura WaitMsg afișează mesajul “Press any key to continue. . .” și așteaptă ca utilizatorul să execute clic pe o tastă. Procedura n-are parametri de intrare. Apelul este următorul:

```
call WaitMsg
```

WriteBin. Procedura WriteBin afișează un întreg fără semn pe 32 de biți în format binar ASCII. Întregul este necesar să fie încărcat în EAX. Biții sunt afișați în grupe de câte 4 biți.

```
mov eax,12346AF9h
call WriteBin
```

În urma invocării se va afișa:

```
0001 0010 0011 0100 0110 1010 1111 1001
```

WriteBinB. Procedura WriteBinB afișează un întreg în format binar pe un octet, cuvânt, sau 32 de biți. Încărcați în EAX întregul și în EBX indicați valoarea de afișat în octeți (1, 2, sau 4). Biții sunt afișați în grupe de câte 4 biți.

```
mov eax,00001234h
mov ebx,TYPE WORD ; 2 octeti inferiori
```

```
call WriteBinB ; displays 0001 0010 0011 0100
```

WriteChar. Procedura *WriteChar* afișează un singur caracter. Încărcați caracterul de afișat (sau codul ASCII al caracterului) în registrul AL.

```
mov al, 'A'
call WriteChar ; displays: "A"
```

WriteDec. Procedura *WriteDec* afișează un întreg fără semn pe 32 de biți în format zecimal. Încărcați întregul în EAX.

```
mov eax, 295
call WriteDec ; displays: "295"
```

WriteHex. Procedura *WriteHex* afișează un întreg pe 32 de biți în format hexazecimal. Introduceți zerouri, dacă este necesar. Încărcați întregul în EAX.

```
mov eax, 7FFFh
call WriteHex ; displays: "00007FFF"
```

WriteHexB. Procedura *WriteHexB* afișează un întreg de un byte, word, sau doubleword în format hexazecimal. Încărcați în EAX întregul și în EBX indicați valoarea de afișat în octeți (1, 2, sau 4).

```
mov eax, 7FFFh
mov ebx, TYPE WORD ; 2 bytes
call WriteHexB ; displays: "7FFF"
```

WriteInt. Procedura *WriteInt* afișează un întreg cu semn pe 32 de biți în format zecimal. Este necesar să introduceți semnul. Încărcați întregul în EAX.

```
mov eax, 216543
call WriteInt ; displays: "+216543"
```

WriteString. Procedura *WriteString* afișează un șir, finalizat cu un octet nul. Încărcați offset-ul șirului în registrul EDI.

```
.data
prompt BYTE "Enter your name: ", 0
.code
mov edi, OFFSET prompt
call WriteString
```

WriteWindowsMsg. Procedura *WriteWindowsMsg* afișează un șir care conține cele mai recente erori generate de MS-Windows, la invocarea funcțiilor de sistem.

```
call WriteWindowsMsg
```

Un exemplu de mesaj:

```
Error 2: The system cannot find the file specified.
```

2.7.7 Instrucțiuni de salt și ciclare

a) Instrucțiunea de salt necondiționat JMP

Forma generală :

JMP operand

unde, operand este adresa de salt necondiționat. Există următoarele tipuri de instrucțiuni JMP:

- de tip SHORT - când operandul specifică o adresă în domeniul -128÷ +127 față de (IP) actualizat
- de tip NEAR - operandul specifică o adresă din același segment de cod;
- de tip FAR - operandul specifică o adresă din alt segment de cod.

b) Instrucțiuni de salt condiționat

Aceste instrucțiuni implementează salturile condiționate de indicatorii de condiție.

Forma generală:

Jcond operand

unde:

- *cond* este condiția de salt și este reprezentată de una sau două litere (vezi tabelul de mai jos);
- *operand* este un offset cuprins între -128 și 128.

Dacă condiția este îndeplinită are loc saltul la adresa dată de operand, dacă nu - se continuă în secvență.

Se observă că există 2 categorii de instrucțiuni pentru 'mai mic' și 'mai mare', cele care conțin cuvintele 'above' sau 'bellow' și cele care conțin cuvintele 'less' sau 'greater'. Primele se folosesc în situația comparării a două valori fără semn, iar ultimele în situația comparării a două valori cu semn.

Fie secvențele de program:

```
mov ax, 0FFFFh
mov bx, 2
cmp ax, bx
ja alfa
```

și

```
mov ax, 0FFFFh
mov bx, 2
cmp ax, bx
jg alfa
```

în care se compară pe cuvânt 0FFFFh și 2.

Se observă că $(AX) > (BX)$ dacă cele două valori se consideră reprezentate fără semn și că $(AX) < (BX)$ dacă cele două valori se consideră cu semn. (-2 este mai mic decât 2). Ca atare în primul caz saltul la eticheta alfa are loc, pe când în cel de-al doilea caz nu are loc.

Fiecare mnemonică din tabel se referă la inițialele cuvintelor următoare, ce indică condiția în limba engleză: Above (peste, mai mare), Below (sub, mai mic), Equal (egal), Not (nu), Greater (mai mare), Less (mai mic), Carry (transport), Zero, Overflow (depășire de capacitate), Parity (PEven - paritate pară, POdd - paritate impară), Sign (semn).

Instrucțiune (mnemonică)	Condiție de salt	Interpretare
JE, JZ	ZF = 1	Zero, Equal
JL, JNGE	SF \neq OF	Less, Not Greater or Equal
JLE, JNG	SF \neq OF sau ZF = 1	Less or Equal, Not Greater
JB, JNAE, JC	CF = 1	Below, Not Above or Equal, Carry
JBE, JNA	CF = 1 sau ZF = 1	Below or Equal, Not Above
JP, JPE	PF = 1	Parity, Parity Even
JO	OF = 1	Overflow
JS	SF = 1	Sign
JNE, JNZ	ZF = 0	Not Zero, Not Equal
JNL, JGE	SF = OF	Not Less, Greater or Equal
JNLE, JG	SF = OF și ZF = 0	Not Less or Equal, Greater
JNB, JAE, JNC	CF = 0	Not Below, Above or Equal, Not Carry
JNBE, JA	CF = 0 și ZF = 0	Not Below or Equal, Above
JNP, JPO	PF = 0	Not Parity, Parity Odd
JNO	OF = 0	Not Overflow
JNS	SF = 0	Not Sign

(Exemplu de citire: JNBE = jump if not below or equal, salt (J) dacă nu (N) e mai mic (B) sau egal (E)).

c) Instrucțiunea JCXZ (JUMP if CX is Zero)

Instrucțiunea realizează salt la eticheta specificată dacă conținutul registrului CX este zero. Forma generală:

JCXZ eticheta**JECXZ eticheta****JRCXZ eticheta**

unde *eticheta* este o eticheta aflata in domeniul -128 si 127 fata de (IP).

Exemple:

```

mov edx,0A523h
cmp edx,0A523h
jne L5 ; jump not taken
je L1 ; jump is taken

mov bx,1234h
sub bx,1234h
jne L5 ; jump not taken
je L1 ; jump is taken

mov cx,0FFFFh
inc cx
jcxz L2 ; jump is taken

xor ecx,ecx
jecxz L2 ; jump is taken

```

Exemple, comparări cu semn:

```

mov edx,-1
cmp edx,0
jnl L5 ; jump not taken (-1 >= 0 is false)
jnle L5 ; jump not taken (-1 > 0 is false)
jl L1 ; jump is taken (-1 < 0 is true)

mov bx,+32
cmp bx,-35
jng L5 ; jump not taken (+32 <= -35 is false)
jnge L5 ; jump not taken (+32 < -35 is false)
jge L1 ; jump is taken (+32 >= -35 is true)

mov ecx,0
cmp ecx,0
jg L5 ; jump not taken (0 > 0 is false)
jnl L1 ; jump is taken (0 >= 0 is true)

mov ecx,0
cmp ecx,0
jl L5 ; jump not taken (0 < 0 is false)
jng L1 ; jump is taken (0 <= 0 is true)

```

Exemplu, salt la etichetă dacă toți biții (2, 3 și 7) sunt setați în 1:

```

mov al,status
and al,10001100b ; mask bits 2,3,7
cmp al,10001100b ; all bits set?
je ResetMachine ; yes: jump to label

```

d) Instrucțiunea LOOP

Forma generală:

LOOP eticheta

Are ca efect:

```

    ecx ← ecx - 1
    dacă ecx = 0 atunci
        [IP] ← [IP] + D8

```

adică se decrementează ECX (CX și RCX în modurile pe 16 și 64 biți respectiv) și dacă acesta este diferit de zero se sare la eticheta specificată, în caz contrar se continuă cu instrucțiunea următoare. D8 este un deplasament pe 8 biți și reprezintă diferența între offset-ul instrucțiunii următoare instrucțiunii LOOP și offset-ul etichetei. Se utilizează și notațiile:

- instrucțiunea LOOPD utilizează registrul ECX ca registru contor;
- instrucțiunea LOOPW utilizează registrul CX ca registru contor.

Ex: Suma celor n octeți de la adresa sir.

```

.data
sir byte 7, 9, 15, 25, -18, 33, 11
n equ LENGTHOF sir
suma byte ?
.code

```

```

    xor eax, eax
    mov ecx, n
    xor esi, esi
    repeta:
        add al, sir[esi]
        inc esi
    LOOP repeta
    mov suma, al

```

e) Instrucțiunea LOOPZ/LOOPE (LOOP While Zero/Equal)

Forma generală:

LOOPZ eticheta

sau

LOOPE eticheta

Semnificația:

```

    cx ← cx - 1
    dacă cx > 0 și ZF = 1 atunci
        [IP] ← [IP] + D8

```

Se decrementează cx (ECX, RCX) și dacă acesta este diferit de zero și indicatorul ZF este 1 (rezultatul ultimei operații aritmetice a fost zero) se sare la eticheta specificată.

f) Instrucțiunea LOOPNZ/LOOPNE (Loop While Not Zero/Not Equal)

Forma generală:

LOOPNZ eticheta

sau

LOOPNE eticheta

Semnificația:

```

    cx ← cx - 1
    dacă cx > 0 și ZF = 0 atunci
        [IP] ← [IP] + D8

```

Efectul este că se ciclează cât timp rezultatul ultimei operații aritmetice este diferit de zero, dar nu de mai multe ori cât este conținutul inițial a lui CX (ECX, RCX).

2.7.8 Instrucțiuni de deplasare (SHL, SAL, SHR, SAR) și de rotație (ROL, RCL, ROR, RCR).

Acest grup de instrucțiuni realizează operații de deplasare și de rotație la nivel de bit. Instrucțiunile au doi operanzi: primul este operandul propriu-zis, iar al doilea este numărul de biți cu care se deplasează sau se rotește primul operand. Ambele operații se pot face la dreapta sau la stânga.

Deplasarea înseamnă transferul tuturor biților din operand la stânga/dreapta, cu completarea unei valori fixe în dreapta/stânga și cu pierderea biților din stânga/dreapta. Deplasarea cu un bit la stânga este echivalenta cu înmulțirea operandului cu 2, iar deplasarea la dreapta, cu împărțirea operandului la 2.

Rotație înseamnă transferul tuturor biților din operand la stânga/dreapta, cu completarea în dreapta/stânga cu biții care se pierd în partea opusă.

Ambele operații se fac cu modificarea bistabilului CF, care participă la operațiile de rotație.

Forma generală a instrucțiunilor este:

OPERATIE operand, contor

în care *operand* este un registru sau o locație de memorie de 8 sau 16 biți, iar *contor* (numărul de biți) este fie o constantă, fie registrul CL, care conține numărul de biți cu care se deplasează/rotește operandul.

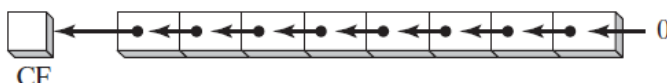
La operațiile de deplasare, se modifică toate flag-urile conform rezultatului, în afară de AF, care este nedefinit. La operațiile de rotație, se modifică numai CF și OF.

La instrucțiunile de deplasare, se consideră deplasări logice și aritmetice, care se pot utiliza după natura operandilor.

a) Instrucțiunea SHL/SAL (Shift Logic/Arithmetic Left)

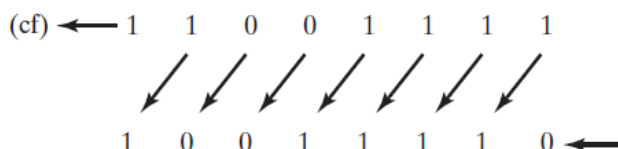
Are forma generală:

SHL/SAL operand, contor



Deși există două mnemonice (SHL și SAL), în fapt este vorba de o unică instrucțiune. Bitul cel mai semnificativ al operandului trece în CF, după care toți biții se deplasează la stânga cu o poziție. Operația se repetă de atâtea ori de cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

Dacă deplasăm valoarea binară 11001111 în stânga cu un bit, după deplasare va deveni 10011110.



Structura instrucțiunii (similar și SHR, SAL, SAR, ROR, ROL, RCR, RCL) este următoarea:

SHL reg, imm8

SHL mem, imm8

SHL reg, CL

SHL mem, CL

Exemple:

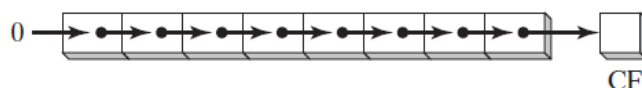
```
mov bl, 8Fh ; BL = 10001111b
shl bl, 1 ; CF = 1, BL = 00011110b
```

```
mov al, 10000000b
shl al, 2 ; CF = 0, AL = 00000000b
```

```
mov dl, 10 ; before: 00001010
shl dl, 2 ; after: 00101000
```

b) Instrucțiunea SHR (Shift Logic Right)

Are forma generală:

SHR *operand, contor*

Bitul cel mai puțin semnificativ din *operand* trece în CF, după care se deplasează toți biții cu o poziție la dreapta (împărțire la 2). Faptul că operația de împărțire se execută fără semn înseamnă că se completează cu un bit 0 dinspre stânga. Operația se repetă de atâtea ori cât este valoarea lui contor (0-255 sau conținutul registrului CL).

Exemple:

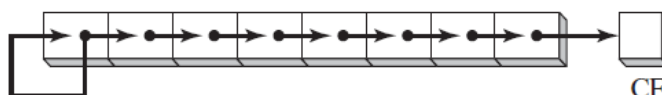
```
mov al,0D0h ; AL = 11010000b
shr al,1 ; AL = 01101000b, CF = 0
```

```
mov al,00000010b
shr al,2 ; AL = 00000000b, CF = 1
```

c) Instrucțiunea SAR (Shift Arithmetic Right)

Are forma generală:

SAR *operand, contor*



Bitul de semn rămâne nemodificat. Bitul cel mai puțin semnificativ din *operand* trece în CF, după care se deplasează toți biții cu o poziție la dreapta. Faptul că operația de deplasare se execută cu semn înseamnă că se completează toți biții eliberați cu bitul de semn. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

Exemple:

```
mov al,0F0h ; AL = 11110000b (-16)
sar al,1 ; AL = 11111000b (-8), CF = 0
```

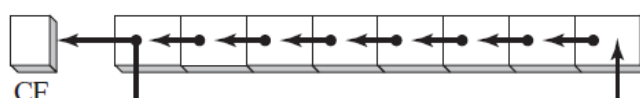
```
mov dl,-128 ; DL = 10000000b
sar dl,3 ; DL = 11110000b
```

```
mov eax,-128 ; EAX = ????FF80h
shl eax,16 ; EAX = FF800000h
sar eax,16 ; EAX = FFFFFFFF80h
```

d) Instrucțiunea ROL (Rotate Left)

Are forma generală:

ROL *operand, contor*

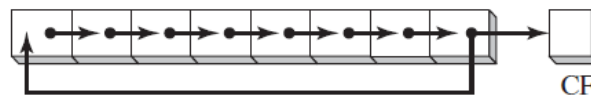


Bitul cel mai semnificativ din *operand* trece atât în CF, cât și în bitul cel mai puțin semnificativ din *operand*, după ce toți biții acestuia s-au deplasat la stânga cu o poziție. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

```
mov al,40h ; AL = 01000000b
rol al,1 ; AL = 10000000b, CF = 0
rol al,1 ; AL = 00000001b, CF = 1
rol al,1 ; AL = 00000010b, CF = 0
```

e) Instrucțiunea ROR (Rotate Right)

Are forma generală:

ROR *operand, contor*

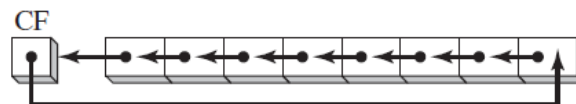
Bitul cel mai puțin semnificativ din *operand* trece atât în CF, cât și în bitul cel mai semnificativ din operand, după ce toți biții acestuia s-au deplasat la dreapta cu o poziție. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

```
mov al,01h ; AL = 00000001b
ror al,1 ; AL = 10000000b, CF = 1
ror al,1 ; AL = 01000000b, CF = 0

mov al,00000100b
ror al,3 ; AL = 10000000b, CF = 1
```

f) Instrucțiunea RCL (Rotate Left through Carry)

Are forma generală:

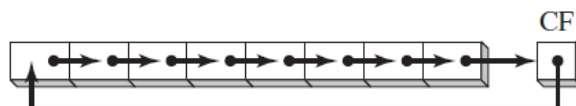
RCL *operand, contor*

Bitul cel mai semnificativ din *operand* trece în CF, se deplasează toți biții din operand cu o poziție la stânga, iar CF inițial trece în bitul cel mai puțin semnificativ din operand. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

```
clic ; CF = 0
mov bl,88h ; CF,BL = 0 10001000b
rcl bl,1 ; CF,BL = 1 00010000b
rcl bl,1 ; CF,BL = 0 00100001b
```

f) Instrucțiunea RCR (Rotate right through Carry)

Are forma generală:

RCR *operand, contor*

Bitul cel mai puțin semnificativ din *operand* trece în CF, se deplasează toți biții din operand cu o poziție la dreapta, iar CF inițial trece în bitul cel mai semnificativ din operand. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

```
stc ; CF = 1
mov ah,10h ; AH, CF = 00010000 1
rcr ah,1 ; AH, CF = 10001000 0
```

Instrucțiunile SHLD/SHRD (shift left double, shift right double)

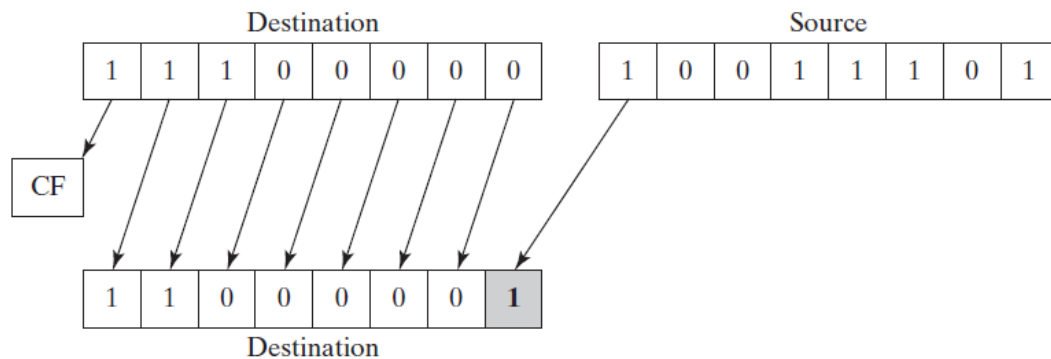
Forma generală:

SHLD *dest, sursa, contor***SHRD** *dest, sursa, contor*

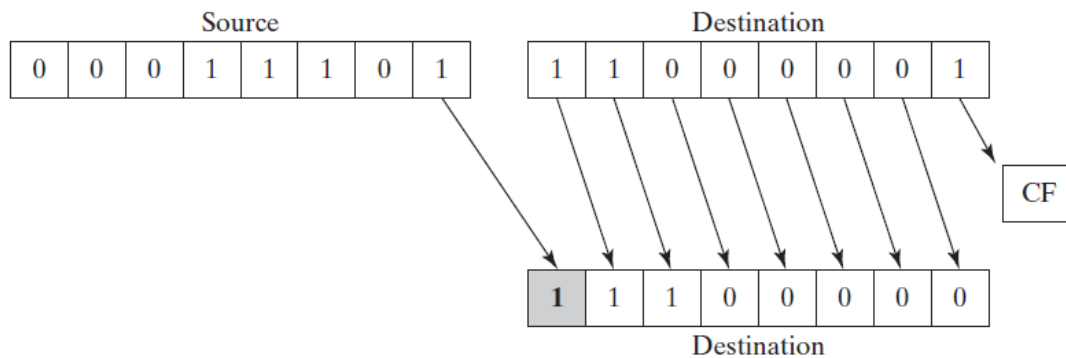
Structura instrucțiunii:

```
SHLD reg16,reg16,CL/imm8
SHLD mem16,reg16,CL/imm8
SHLD reg32,reg32,CL/imm8
SHLD mem32,reg32,CL/imm8
```

Deplasarea stânga se petrece după următoarea schemă:



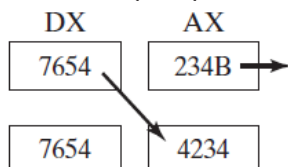
Iar deplasarea dreapta după următoarea schemă:



Exemple:

```
.data
wval WORD 9BA6h
.code
mov ax,0AC36h
shld wval,ax,4 ; wval = BA6Ah
```

```
mov ax,234Bh
mov dx,7654h
shrd ax,dx,4
```



2.7.9 Instrucțiuni logice

Instrucțiunile logice realizează funcțiile logice de bază, pe 8, 16, 32 sau 64 biți. Operațiile se fac la nivel de bit.

a) Instrucțiunea NOT (Not)

Forma generală :

NOT dest

în care *dest* poate fi un registru sau o locație de memorie. Instrucțiunea provoacă negarea tuturor biților

operandului, adică se face complementul față de unu.

```
mov al,11110000b
not al ; AL = 00001111b
```

b) Instrucțiunea AND (And)

Forma generală :

AND dest, sursa

în care *dest* poate fi un registru sau o locație de memorie, iar *sursa* un registru, o locație de memorie sau o constantă. Instrucțiunea depune în *dest* și-logic la nivel de bit între *dest* și *sursa*.

Indicatori afectați: SF, ZF, PF, CF=0, OF=0, AF nedefinit. Structura instrucțiunii:

```
AND reg, reg
AND reg, mem
AND reg, imm
AND mem, reg
AND mem, imm
```

Valoarea imediată *imm* nu poate depăși mărimea de 32 biți.

```
mov al,10101110b
and al,11110110b ; result in AL = 10100110
```

c) Instrucțiunea OR (Or)

Forma generală :

OR dest, sursa

în care *dest* poate fi un registru sau o locație de memorie, iar *sursa* un registru, o locație de memorie sau o constantă. Instrucțiunea depune în *dest* sau-logic la nivel de bit între *dest* și *sursa*.

Indicatori afectați: SF, ZF, PF, CF=0, OF=0, AF nedefinit.

```
mov al,11100011b
or al,00000100b ; result in AL = 11100111
```

d) Instrucțiunea XOR (Exclusive Or)

Forma generală :

XOR dest, sursa

în care *dest* poate fi un registru sau o locație de memorie, iar *sursa* un registru, o locație de memorie sau o constantă. Instrucțiunea depune în *dest* xor-logic la nivel de bit între *dest* și *sursa*.

Indicatori afectați: SF, ZF, PF, CF=0, OF=0, AF nedefinit. Tabelul de adevăr este următorul:

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

e) Instrucțiunea TEST (Test)

Forma generală :

TEST dest, sursa

în care *dest* poate fi un registru sau o locație de memorie, iar *sursa* un registru, o locație de memorie sau o constantă. Instrucțiunea realizează and-logic la nivel de bit între *dest* și *sursa* și nu modifică destinația, cu poziționarea indicatorilor.

Indicatori afectați: SF, ZF, PF, CF=0, OF=0, AF nedefinit.

Această instrucțiune este deseori utilizată pentru a testa starea unui bit (biți).

Exemplu. Fie este necesar să testăm starea bitului 5:

```
test al,00100000b; test bit 5
```

Dacă ZF=0, atunci bitul 5 este egal cu 1, iar dacă ZF=1, bitul 5 egal cu 0.

f) Instrucțiunile BSF, BSR (Bit scan forward, reverse)

Forma generală :

BSF dest,sursaBSF *reg16,r/m16*BSF *reg32,r/m32*

- Instrucțiunea scanează biții operandului sursă, începând cu bitul 0 (BSR-15/31) până la bitul 15/31 (BSR-0), pentru a găsi primul bit de 1.
- Dacă se întâlnește bitul de 1, flag-ul ZF este setat în 0, iar în operandul destinație este încărcat indexul (poziția) primului bit de 1.
- În cazul în care nici un bit de 1 nu este găsit, flag-ul ZF este setat în 1.

Exemplu:

```

        mov    bx,0002h    ;bx=0000 0010b
        ...
    bsf     cx,bx    ;cx=0001h
    jz      null
        ...
null:

```

Instrucțiunile BT, BTC, BTR, BTS (Bit tests)

Forma generala :

BT dest,sursaBT *r/m16,imm8*BT *r/m16,r16*BT *r/m32,imm8*BT *r/m32,r32*

Instrucțiunea BT copie bitul *n*, specificat în sursă, în flag-ul Carry. Operandul destinație conține valoarea cu bitul căutat, iar operandul sursă conține poziția bitului căutat.

Instrucțiunea BTC copie bitul *n*, în flag-ul Carry și compementează bitul *n* în operandul destinație.

Instrucțiunea BTR copie bitul *n*, în flag-ul Carry și setează bitul *n* în 0 în operandul destinație.

Instrucțiunea BTS copie bitul *n*, în flag-ul Carry și setează bitul *n* în 1 în operandul destinație.

```

        mov    ebx,01001100h
        bt     ebx,8    ;testarea bitului 8 și setarea cf în 1
        jc     m1      ;salt la m1, dacă valoarea bitului este 1

```

Exemple programe

{Căutare secvențială} Să se scrie un program pentru căutarea primului blanc din șirul începând de la adresa **sir**. La ieșirea din program **eax** va conține valoarea 0 dacă șirul nu conține blancuri, altfel va conține valoarea poziției din sir a primului blanc găsit. Se presupune că șirul **sir** are 1 caractere.

```

.data
sir    DB    'Acesta este un sir!'
l      EQU   sizeof sir

.code

        mov    ecx,l
        mov    esi,-1
        mov    al,''
urm:    inc     esi
        cmp    al,sir[esi]
        loopne urm
        jne    nu_gasit
        mov    eax,l
        sub    eax,ecx

```

```

    jmp    iesire
nu_gasit: mov    eax,0
iesire:  nop

```

2.8 Instrucțiuni pentru controlul procesorului

Sunt instrucțiuni care controlează anumite funcții ale procesorului, ce acționează fie prin intermediul unor indicatori de control (sau registre de control), fie prin introducerea unor stări sau semnale necesare pentru sincronizarea cu evenimentele externe.

Exemple:

CMC ;complementarea valorii indicatorului CF
 CLC ;poziționarea pe 0 a indicatorului CF
 STC ;poziționarea pe 1 a indicatorului CF
 NOP ;Nici o operație, dar consumă 3 perioade de ceas.
 CLD ;poziționarea pe 0 a indicatorului DF
 STD ;poziționarea pe 1 a indicatorului DF
 CLI ;poziționarea pe 0 a indicatorului IF, dezactivare întreruperi mascabile
 STI ;poziționarea pe 1 a indicatorului IF
 HLT ;Oprire microprocesor până la RESET, NMI, sau INT (dacă sunt activate)
 WAIT ;așteptare până când vine semnalul exterior test=0
 ESC ;operație destinată coprocesorului
 LOCK ;prefix care activează semnalul extern /lock, astfel că microprocesorul anunță
 ;că nu va răspunde la o cerere de cedare a controlului magistralelor.

2.9 Instrucțiuni pentru lucrul cu șiruri

În afară de tipurile de bază amintite mai sus, există și posibilitatea efectuării unor operații de transfer, sau operații aritmetice și logice cu șiruri de date (cu informații aflate în zone continue de memorie). Operațiile pe șiruri pot fi efectuate individual, pentru fiecare cuvânt din șir, sau automat - cu repetare, numărul de repetări al instrucțiunii fiind dictat de conținutul unui registru contor.

Operațiile tipic efectuate sunt:

- transferul unui șir din zonă sursă în zonă destinație
- comparare între două șiruri
- căutarea unei valori într-un șir
- încărcarea acumulatorului cu elementele unui șir.
- citirea unui șir de la un port de intrare
- scrierea unui șir la un port de ieșire

Exemple :

Instrucțiunile MOVSB (Move (copy) bytes)
MOVSX (Move (copy) words)
MOVSD (Move (copy) doublewords)

Transfer pe 8 (16,32) biți, din zona de memorie indicată de ESI, în zona de memorie indicată de registrul EDI. După transferul primului byte (word, doubleword), dacă flag-ul DF=0, se petrece autoincrementarea ESI←ESI+1; EDI←EDI+1 (decrementare pentru DF=1).

În operații cu șiruri sunt utilizate prefixe de repetare:

REP	Repetare până ecx>0
REPZ, REPE	Repetare până ZF=1 și ecx>0
REPNZ, REPNE	Repetare până ZF=0 și ecx>0

Exemplu. Fie dat să copiem 20 de cuvinte duble din șirul sursă **source** în șirul destinație **target**:

```

.data
source DWORD 20 DUP(0FFFFFFFFh)
target  DWORD 20 DUP(?)

```

```
.code
cld                                ; direction = forward
mov ecx,LENGTHOF source ; setam contorul REP
mov esi,OFFSET source ; incarcam ESI cu adresa sourcei
mov edi,OFFSET target ; incarcam EDI cu adresa destinației
rep movsd                        ;copiem cuvinte duble
```

Instrucțiunile CMPSB (Compare bytes) CMPSW (Compare words) CMPSD (Compare doublewords)

Comparare pe 8 (16,32) biți, din zona de memorie indicată de ESI, cu zona de memorie indicată de registrul EDI. După compararea primului byte (word, doubleword), dacă flag-ul DF=0, se petrece autoincrementarea ESI←ESI+1; EDI←EDI+1 (decrementare pentru DF=1).

Exemple:

```
.data
source DWORD 1234h
target DWORD 5678h
.code
mov esi,OFFSET source
mov edi,OFFSET target
cmpsd                        ; compare doublewords
ja L1                        ; jump if source > target
```

Dacă comparăm cuvinte multiple:

```
mov esi,OFFSET source
mov edi,OFFSET target
cld ; direction = forward
mov ecx,LENGTHOF source ; repetition counter
repe cmpsd ; repeat while equal
```

Prefixul REPE repetă compararea, incrementând ESI și EDI în mod automat, până când ECX =0 sau o pereche de cuvinte duble nu va fi egală.

Instrucțiunile SCASB (SCAS- Scans a string) SCASW SCASD

Instrucțiunile compară valoarea din AL/AX/EAX cu byte, word sau doubleword din zona de memorie indicată de EDI. Instrucțiunile sunt utile la căutarea unui singur element într-un șir.

Exemple:

```
.data
alpha BYTE "ABCDEFGH",0
.code
mov edi,OFFSET alpha ; incarcam EDI cu adresa
                        ;sirului de scanat
mov al,'F'            ; cautam litera F
mov ecx,LENGTHOF alpha ; setam registrul contor
cld                    ; direction = forward
repne scasb ; repetam pana nu este egal
jnz quit ; iesire daca litera nu a fost gasita
```

Instrucțiunile STOSB (STOS- Store string data) STOSW STOSD

Instrucțiunile încarcă valoarea din AL/AX/EAX , în memorie cu offset-ul indicat de EDI. Incrementarea se petrece conform flag-ului DF (DF=0- incrementarea, DF=1- decrementarea).

Exemplu. Șirul `string1` este completat cu valoarea 0FFh.

```
.data
Count = 100
string1 BYTE Count DUP(?)
.code
mov al,0FFh ; valoarea de de incarcat
mov edi,OFFSET string1 ; EDI cu adresa sirului
mov ecx,Count ; numarul de elemente ale sirului
cld ; direction = forward
rep stosb ; copierea AL in string1
```

Instrucțiunile LODSB (LODS- Load Accumulator from String)

LODSW

LODSB

Instrucțiunile încarcă valoarea din byte, word sau doubleword din memorie idicat de ESI, în AL/AX/EAX respectiv. Instrucțiunile sunt utile la căutarea unui singur element într-un șir.

Exemplu: Multiplicarea fiecărui element a unui șir cu o constantă.

```
INCLUDE Irvine32.inc
.data
array DWORD 1,2,3,4,5,6,7,8,9,10 ; test data
multiplier DWORD 10
.code
main PROC
cld ; direction = forward
mov esi,OFFSET array ; sirul sursa
mov edi,esi ; sirul destinatie
mov ecx,LENGTHOF array ; setarea contorului
L1: lodsd ; incarcarea [ESI] in EAX
mul multiplier ; multiplicarea cu constanta
stosd ; copie din EAX in [EDI]
loop L1
exit
main ENDP
END main
```

2.10 Subprograme și macroinstrucțiuni

În general definirea unui subprogram se face cu directiva *PROC* în maniera următoare:

```
nume PROC {NEAR | FAR}
corp
RET {constanta}
nume ENDP
```

Dacă atributul *NEAR* și *FAR* lipsesc, în cazul utilizării definițiilor complete se consideră implicit *NEAR*.

Tehnicile de transfer a parametrilor combină diversele modalități de alegere a tipurilor de locații fizice pentru păstrarea parametrilor transmiși: registre, locații de memorie fixate, codul apelant, stiva procesorului, tehnica blocurilor (tabelelor) de parametri, cu ceea ce se transmite efectiv referitor la un anumit parametru: adresa sau valoarea acestuia.

Instrucțiunea CALL (apel de procedura)

Poate apărea sub una din formele:

```
CALL nume_proc
CALL NEAR PTR nume_proc
```


CALL FAR PTR nume_proc

Tipul apelului poate fi dedus din tipul procedurii (primul caz) sau specificat explicit prin NEAR și FAR. Tipul apelului trebuie să coincidă cu tipul procedurii și cu tipul instrucțiunii RETURN din interiorul procedurii.

Instrucțiunea RET (RETURN)

Forma generală:

RET [n]

unde **n** este o constantă întreagă opțională.

Dacă instrucțiunea RET este de tip NEAR semnificația sa este:

$[IP] \leftarrow SS: [[SP] + 1: [SP]]$

$[SP] \leftarrow [SP] + 2$

$[[SP] \leftarrow [SP] + n]$

adică se reface (IP) prin copierea conținutului vârfului stivei și incrementarea cu 2 a lui (SP). Dacă în instrucțiunea RET apare și constanta **n** atunci această constantă se adună la (SP), adică se descarcă stiva.

Exemplu:

```
.data
a   DWORD 55555h
b   DWORD 77777h
s   DWORD ?
.code
...
; încarcă primul număr în DX:AX
mov ax,WORD PTR a
mov dx,WORD PTR [a+2]
; încarcă al doilea număr în DI:SI
mov si,WORD PTR b
mov di,WORD PTR [b+2]
; încarcă adresa rezultatului în BX
mov bx,OFFSET s
; apelează procedura
call pro_ad
...
; codul procedurii
pro_ad PROC NEAR
add ax,si
adc dx,di
mov [bx],ax
mov [bx+2],dx
ret
pro_ad ENDP
...
```

O macroinstrucțiune reprezintă o secvență de cod sursă căreia i se atribuie un nume simbolic, conținutul acestei secvențe putând fi repetat ori de câte ori în cadrul unui program prin simpla referire la numele simbolic respectiv. Utilizarea unei macroinstrucțiuni necesită parcurgerea a doi pași:

1. Definirea macroinstrucțiunii, care se marchează printr-o macro definiție. Aceasta cuprinde o secvență de cod, între directivele MACRO și ENDM. Sintaxa este:

```
nume MACRO {parametrii}
cod
ENDM
```

unde:

- **nume** reprezintă numele simbolic dat macroinstrucțiunii ;

- **parametrii** reprezintă parametrii formali opționali ai macroinstrucțiunii, separați prin virgulă, blankuri sau TAB-uri. La apelul macroinstrucțiunii, acești parametri formali sunt înlocuiți textual cu parametrii actuali.

2. Apelul macroinstrucțiunii, care se realizează printr-un macroapel, cu sintaxa:

nume {argumente}

unde:

- **nume** reprezintă numele simbolic al macroinstrucțiunii apelate;
- **argumente** reprezintă lista parametrilor actuali, separați prin virgulă, blankuri sau TAB-uri.

Apelul macroinstrucțiunii are ca efect includerea textuală a codului din definiția macroinstrucțiunii în corpul programului.

Exemplu : Adunarea a 3 cuvinte cu depunerea rezultatului în ax.

```

        addup MACRO ad1,ad2,ad3
        mov  ax,ad1
        add  ax,ad2
        add  ax,ad3
        ENDM

.....
a  WORD  1
b  WORD  2
cd WORD  3
d  WORD  ?
.....
        addup a,b,cd
        mov  dx,ax
        addup dx,dx,dx
        mov  d,ax
        addup d,dx,cd

```

Pentru definirea unor simboluri în cadrul unei macroinstrucțiuni, care la fiecare apel al macroinstrucțiunii respective vor fi înlocuite cu nume unice de simboluri, gestionate de asamblor, se utilizează directiva LOCAL cu sintaxa:

LOCAL nume {,nume} ...

Directiva LOCAL, dacă este prezentă într-o macrodefiniție, trebuie să urmeze imediat directivei MACRO.

Exemplu: Ridicarea unui număr la o putere.

```

        power MACRO factor,exponent
        LOCAL again, gotzero
        xor  dx,dx
        mov  ax,factor
        mov  cx,exponent
again:   jcxz gotzero
        mul  bx
        loop again
gotzero:
        ENDM

```

3 Structura calculatoarelor

3.1 Componentele funcționale și clasificarea

Noțiunea de la care pornim este aceea de calculator; acesta este un sistem programabil de prelucrare a informației care are două componente esențiale, inseparabile și definitorii: *hardware și software*.

A. Din punct de vedere *hardware*, calculatorul are trei componente funcționale legate într-un mod specific (Figura 3.1).

Blocurile funcționale sunt:

1. *Unitatea centrală de prelucrare* (UCP) are două funcții esențiale:

- prelucrarea datelor;
- controlul activității întregului calculator.

O unitate centrală de prelucrare informației, având funcțiile enunțate mai sus, care coordonează un sistem structurat funcțional ca în Figura 3.1 și care, fizic, se prezintă sub forma unui singur cip (circuit integrat) se numește *microprocesor*. Această accepțiune standard a noțiunii va fi folosită în continuare pentru detalierea conceptelor care stau la baza funcționării întregului calculator.

2. *MEMORIA* este, din punctul de vedere al sistemului pe care îl definim, o secvență de locații pentru stocarea informației.

Fiecare locație este definită prin două entități informaționale:

- *Conținutul*, reprezentat de o înșiruie de cifre binare 0 sau 1 ("biți"); se va observa că nu am folosit noțiunea de "număr binar", pentru că informația stocată într-o locație de memorie poate avea diverse semnificații. Numărul de cifre binare conținute într-o locație depinde de modul în care microprocesorul organizează informația în memorie; mărimea unei locații va fi denumită *formatul memoriei*, exprimat în număr de biți (8 biți). Formatul memoriei nu are nici-o legătură cu *organizarea fizică* a cipurilor de memorie.

Figura 3.1

- *Adresă*, reprezentând numărul de ordine al locației, care permite identificarea sa în cadrul secvenței de locații (există o corespondență biunivocă între fiecare locație de memorie și adresa sa).
- În privința memoriei unui calculator vom folosi câteva noțiuni:
- "*Harta memoriei*", definită ca fiind totalitatea locațiilor de memorie pe care le poate adresa un microprocesor.
- "*Pagini*" și/sau "*segmente*" sunt subdiviziuni *logice* ale hărții memoriei, ale căror dimensiuni, fixe sau dinamice, sunt specifice modului în care un microprocesor anume organizează memoria. Subliniem din nou că aceste moduri de organizare nu au nici-o legătură cu structura fizică a memoriei unui calculator.

3. *Dispozitivele de intrare/ ieșire (I/O)* sunt constituite din circuitele prin care se realizează legătura între calculator și lumea exterioară. O unitate elementară de conversație cu exteriorul poartă numele de "*port de intrare/ieșire*". Între porturi și locațiile din memorie există niște similitudini:

- Porturile sunt în esență tot locații de memorare a informației, adresabile; desigur, informația care se folosește uzual aici este alcătuită din operanzi/rezultate (date).
- Există o "*hartă a porturilor*" care, așa cum vom arăta în capitolele următoare, poate sau nu să facă parte din harta memoriei.

Singura deosebire esențială față de locațiile de memorie este legătura fizică pe care porturile o asigură cu exteriorul; pentru microprocesor, de multe ori, această legătură fizică este transparentă și ne semnificativă.

În sfârșit, componenta hardware a calculatorului comportă un set de legături specifice; acestea se realizează printr-o așa numită "magistrală": un set de conexiuni fizice între blocuri prin care informația care circulă are o semnificație prestabilită. Sistemele la care ne referim au o magistrală unică, ce le caracterizează; din punct de vedere funcțional, există trei componente ale acestei magistrale, individualizate și în Figura 3.1:

1. *Magistrala de date*, bidirecțională, permite circulația datelor (operanzi/rezultate), a instrucțiunilor și chiar a adreselor.

2. *Magistrala de adrese*, unidirecțională, permite microprocesorului să localizeze informația în Memorie sau în Dispozitivele de intrare/ieșire; deci pe această magistrală circulă numai adrese.

3. *Magistrala de control* permite circulația, bidirecțională, a semnalelor de comandă și control de la/la microprocesor, în calitatea sa de Unitate centrală.

B. Din punct de vedere **software**, a doua componentă definitorie a calculatorului, definirea rezultă practic din considerentele anterioare: o serie de programe organizate în moduri specifice.

Prezentarea acestor noțiuni și definirea lor ne permit câteva concluzii care să facă o primă delimitare asupra conceptului de microprocesor așa cum este el înțeles în volumul de față:

- Microprocesorul constituie Unitatea centrală de prelucrare a unui sistem având schema bloc funcțională din Figura 3.1. Important este că el concentrează și funcția de prelucrare și pe cea de comandă.

- Toate celelalte componente ale sistemului nu au putere de decizie. Memoria, de pildă, nu controlează și nici nu e necesar să controleze semnificația informației pe care o deține și modul în care este organizată logic.
- Legătura dintre blocuri este asigurată de o magistrală unică cu trei componente funcționale; pe magistrala de date circulă toate tipurile de informații.
- Funcționarea sistemului se face pe baza unor programe alcătuite din secvențe de instrucțiuni. Acestea sunt citite din memorie de către microprocesor, recunoscute și apoi executate.

Calculatoarele deseori se caracterizează prin :

- a) Viteza de calcul - este o evaluare, determinată analitic sau experimental, a volumului de instrucțiuni (comenzi) executate de calculator într-o unitate de timp. Viteza de calcul se măsoară în milioane de instrucțiuni executate pe secundă (milion instruction per second, MIPS), milioane de instrucțiuni în virgulă mobilă (megaflops, MFLOPS).
- a) Lățimea magistralei de date – lățimea (mărimea) maximală a codului informatic care poate fi prelucrat, păstrat și transferat în calculator, ca o unitate întreagă.
- b) Capacitatea de stocare a memoratoarelor – cantitatea informației codate, concomitent păstrată în memoratoarele calculatorului. Pentru măsurarea capacității de stocare se folosesc următoarele prefixe:
 - 1KiloByte = 2^{10} Byte $\approx 10^3$ Byte
 - 1MegaByte = 2^{20} Byte $\approx 10^6$ Byte
 - 1GigaByte = 2^{30} Byte $\approx 10^9$ Byte
 - 1TerraByte = 2^{40} Byte $\approx 10^{12}$ Byte
 - 1PetaByte = 2^{50} Byte $\approx 10^{15}$ Byte

Clasificarea calculatoarelor conform caracteristicilor menționate este neactuală.

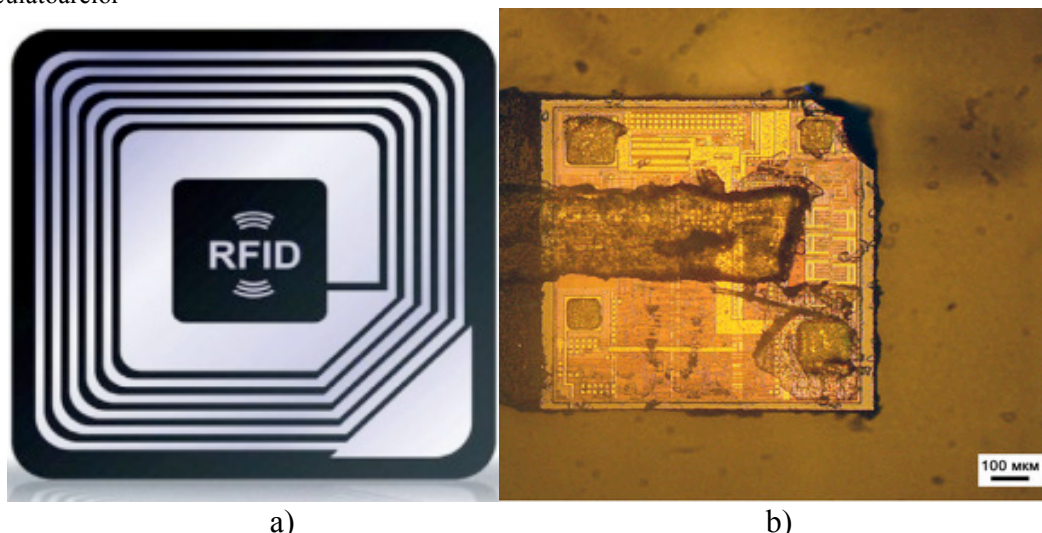
Prezentăm câteva clasificări moderne.

Industria modernă produce o mare varietate de calculatoare. Din această varietate, vom clasifica orientativ calculatoarele (E. Tanenbaum "Organizarea structurată a calculatoarelor"):

- circuitul integrat, sau calculatoare "one-off", domeniul de utilizare – de ex. felicitările (cărți poștale);
- calculatoare integrate (microcontrolere) - ceasuri, mașini, diferite dispozitive;
- console de jocuri – jocuri la domiciliu;
- calculatoare personale PC – variantele desktop și laptop-uri;
- servere - servere de rețea;
- grupe (clustere) de stații de lucru (COW- Cluster Of Workstations) - multicalculatoare conectate în rețele;
- Mainframe - prelucrarea a bazelor de date într-o bancă.

Circuite integrate

Ele mai sunt numite calculatoare "one-off". Aceste circuite pot fi lipite pe cărți poștale și interpretează melodii cu tematica specifică anumitor sărbători / aniversari, de tipul «Happy Birthday». Probabil, cea mai semnificativă realizare în acest domeniu a fost apariția circuitelor RFID (Radio Frequency Identification — Tehnologia identificării prin radiofrecvență). Această tehnologie presupune stocarea informațiilor nu prin codurile de bare, ci prin intermediul unor cipuri electronice integrate, de ex. în etichete, ecusoane, ambalaje de marfă, corpurile animalelor etc. Aceste informații, ce reprezintă un cod unic din 128 biți, pot fi citite (de la câțiva centimetri până la sute de metri) prin unde radio. Dimensiunea acestor circuite este mai mică de 0,5mm (figura 3.2), costul fiind de câțiva cenți. Circuitele nu utilizează surse de alimentare și pot stoca informația mult timp.



a) b)
Figura 3.2 - Circuitul RFID -a, b – fără antenă

Informații detaliate le puteți găsi pe site-ul www.rfid.org.

Microcontrolere

La modul general un microcontroler este, actualmente, o structură electronică destinată controlului (destul de evident!) unui proces sau, mai general, unei interacțiuni caracteristice cu mediul exterior, fără să fie necesară intervenția operatorului uman. El reprezintă un microcircuit care încorporează o unitate centrală (CPU) și o memorie împreună cu resurse care-i permit interacțiunea cu mediul exterior. Toate aplicațiile în care se utilizează microcontrolere fac parte din categoria așa ziselor sisteme încapsulate-integrate (“embedded systems”), la care existența unui sistem de calcul încorporat este (aproape) transparentă pentru utilizator.



Printre multe domenii, unde utilizarea lor este practic un standard industrial, se pot menționa:

în industria de automobile (controlul aprinderii/motorului, climatizare, diagnoză, sisteme de alarmă, etc.), în așa zisa electronică de consum (sisteme audio, televizoare, camere video și videocasetofoane, telefonie mobilă, GPS-uri, etc.), în aparatura electrocasnică (mașini de spălat, frigidere, cuptoare cu microunde, aspiratoare), în controlul mediului și climatizare (sere, locuințe, hale industriale), în industria aerospațială, în mijloacele moderne de măsurare (aparate de măsurare, senzori și traductoare inteligente), la realizarea de periferice pentru calculatoare, în medicină, ș.a.

Ca un exemplu din industria de automobile, unde numai la nivelul anului 1999, un BMW seria 7 utiliza 65 de microcontrolere, iar un Mercedes din clasa S utiliza 63 de microcontrolere; iar un avion peste 200. Practic, este foarte greu de găsit un domeniu de aplicații, în care să nu se utilizeze microcontrolerele.

Console de jocuri. O consolă de jocuri este un sistem dedicat jocurilor video, ce reprezintă de fapt un calculator interactiv pentru distracții. Deseori constă din 2 unități: un controller - cu ajutorul căruia, utilizatorul poate introduce date sau interacționa cu obiectele de pe ecran și un bloc ce conține un procesor, RAM, și un coprocesor pentru audio-video, încorporate într-o carcasă la care se conectează televizorul și controlerul. Prezentăm caracteristicile principale ale Sony PlayStation4. Noua consolă are un procesor AMD Jaguar cu 8 nuclee și un GPU AMD Radeon, cu o viteză de calcul de 1,84 teraflopi, alături de memorie RAM GDDR5 de 8GB și spațiu de stocare HDD până 640 GB. Specificațiile PS4 includ și Blu-ray drives cu un volum de 250 GB, alături de conectivitate USB 3.0, Bluetooth 4.0, HDMI,

Calculatoare personale

Calculatoare personale (PC) se divizează în 2 grupe: variantele desktop și portabile (laptop, notebook, palmtop (PDA)). În structura lor, de regulă, intră: microprocesoare, module de memorie de gigabytes, hard discuri de terabytes, CD-ROM/DVD drives, modemuri, plăci video, audio, de rețea, monitoare ș.a., sisteme de operare complexe instalate.

Servere

Un server este o un calculator, care operează continuu în rețeaua sa și așteaptă solicitări din partea altor calculatoare din rețea, pentru a asigura accesul la toată paleta de forme de conectare și servicii. Multe componente de hardware sunt identice cu cele ce le găsim într-un calculator personal. Totuși serverele rulează sisteme de operare și programe specializate care sunt diferite față de cele folosite pe calculatoare personale.

Serverele deservește resurse hardware care sunt partajate și pot uneori fi comandate de către calculatoarele-client, cum ar fi imprimante (atunci serverul se numește *print server*) sau sisteme de fișiere (atunci el se numește *file server*). Această partajare permite un acces și o securitate mai bune. Cu toate că serverele pot fi construite, din comoditate, din componente obișnuite de calculatoare, este necesar ca, pentru operații rapide și de mare amploare, serverele să folosească configurații hardware optimizate pentru aceste cerințe, Intel produce microprocesoare specializate pentru servere și stații de lucru - Intel Xeon. Cu toate că serverele oferă mult spațiu pe disc, pentru mărirea siguranței în funcționare sunt folosite hard-discuri de capacitate redusă, numeroase, interconectate în mod special.

Folosirea mai multor microprocesoare duce la o mai mare fiabilitate în comparație cu un singur microprocesor. De asemenea se folosesc *Uninterruptible Power Supplies* (UPS-uri) pentru a fi siguri de continuitatea de alimentare cu energie electrică, astfel ca penele din rețeaua publică de curent să nu provoace stricăciuni ireparabile. Diferența majoră între computerele personale și servere nu este partea hardware ci partea de software. Pe servere rulează sisteme de operare care sunt special proiectate pentru acestea. De asemenea ele rulează aplicații special proiectate pentru procesele dorite. În lumea serverelor cele mai populare sistem de operare sunt FreeBSD, Sun Solaris și GNU/Linux – care derivă și sunt asemănătoare cu sistemul de operare UNIX. UNIX a fost o alegere logică și eficientă ca sistem de operare pentru servere.

Grupe (clustere) de stații de lucru (COW- Cluster Of Workstations)

Clusterele constau din zeci, sute, mii de PC-uri sau stații de lucru conectate în rețea prin plăci de rețea de larg consum. Sistemele COW sunt gestionate de soft specializat, ce permite să direcționeze resursele lor pentru a rezolva diferite probleme ingineresti și științifice. Dacă frecvența accesărilor la paginile web-site-ului se estimează la mii, zeci de mii pe secundă, este convenabil ca serverele să fie organizate în clustere de stații de lucru.

Calculatoarele Mainframe

Calculatoarele mainframe sunt calculatoare ce pot exploata volume imense de date și pot suporta lucrul a mii de utilizatori simultan. Un calculator mainframe se distinge mai ales prin capacitatea de stocare și memoria internă. El poate rula ani întregi fără întrerupere. Unele calculatoare pot rula mai multe sisteme de operare simultan, operând astfel ca o mulțime de “mașini virtuale”. Prețul unui astfel de calculator este de ordinul sutelor de mii de dolari. Este solicitat de companiile care vehiculează și prelucrează un volum foarte mare de informație. Principala diferență între supercalculatoare și mainframe este că primele se folosesc pentru operații ce necesită calcule intense, în timp ce mainframe efectuează operații de complexitate redusă asupra unor volume mari de date.

Supercalculatorul posedă resurse hardware și software deosebite. Se utilizează în industria de apărare, în cercetarea științifică, în câteva universități, în industria aeronautică și spațială. Departamentul Energiei SUA deține un supercomputer din lume *The Roadrunner*. Acesta are o putere de calcul de 1 petaflop (10^{15} operații pe secundă, în virgulă mobilă). Ocupă o suprafață de 1100 m² și a fost construit din 700 de procesoare AMD Opteron.

În anul 2013 compania Cray a realizat pentru Departamentul Energiei al SUA un supercomputer cu o putere de calcul de 20 peta operații de secundă (20 petaflops) numit Titan. Astfel, în scurt timp (anul 2025), se va realiza un supercomputer cu o putere de calcul de 10 exaflops (10^{19} flops), care va fi capabil să simuleze activitatea creierului uman.

În studiul arhitecturilor de calcul este foarte utilă existența unei metode de comparare a diferitelor arhitecturi, fără a fi necesară compararea specificațiilor detaliate ale fiecărei arhitecturi. Astfel că arhitecturile de calcul sunt clasificate pe baza unui set mai restrâns de caracteristici.

Clasificarea lui Flynn

Cea mai cunoscută clasificare a arhitecturilor de calcul este cea propusă de Flynn (profesor la Stanford University) în 1966. Această clasificare nu examinează structura explicită a sistemelor ci urmărește fluxul de date și de instrucțiuni prin acestea. Prin flux de instrucțiuni se înțelege secvența de instrucțiuni executată de o mașină sau unitate de execuție; iar prin flux de date se înțelege secvența de date apelate de fluxul de instrucțiuni.

După Flynn arhitecturile de calcul se împart în următoarele patru categorii:

- cu un flux de instrucțiuni și un flux de date (SISD);
- cu un flux de instrucțiuni și mai multe fluxuri de date (SIMD);
- cu mai multe fluxuri de instrucțiuni și un flux de date (MISD);
- cu mai multe fluxuri de instrucțiuni și mai multe fluxuri de date (MIMD).

SISD (Single Instruction Single Data):

Din această categorie fac parte calculatoarele convenționale care execută un singur flux de instrucțiuni asupra unui singur flux de date. Aceste sisteme de calcul se mai numesc și calculatoare von Neumann.

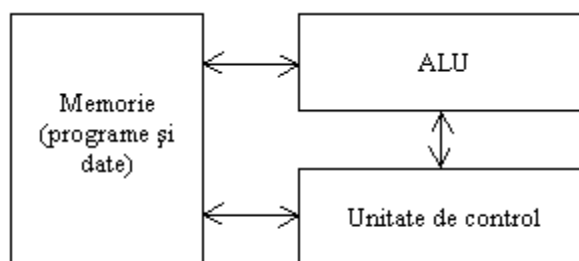


Figura 3.2 Arhitectura von Neumann

Instrucțiunile sunt executate secvențial, însă pot exista suprapuneri între acestea dacă este implementat conceptul de bandă de asamblare (pipeline) – majoritatea sistemelor SISD actuale utilizează conceptul de bandă de asamblare. Calculatoarele SISD pot avea mai multe unități funcționale (ex: coprocesor matematic, procesor grafic, procesor de intrare/ieșire, etc.), însă acestea sunt văzute ca o singură unitate de execuție.

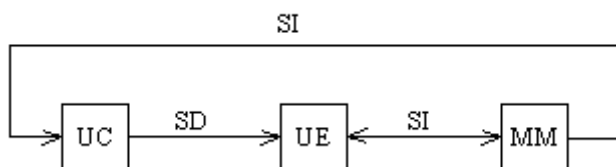


Figura 3.3 Arhitectura SISD

UC – unitate de comandă;

UE – unitate de execuție, element de procesare, procesor;

MM – modul de memorie;

SI – flux (șir) de instrucțiuni;

SD – flux (șir) de date.

Exemple de calculatoare SISD: CDC 6600, CDC 7600, Amdhal 470/6, Cray-1.

SIMD (Single Instruction Multiple Data)

Această categorie de arhitecturi cuprinde sistemele de calcul compuse din mai multe unități de execuție identice aflate sub comanda unei singure unități de control. Unitatea de control transmite același flux de instrucțiuni, simultan, tuturor unităților de execuție. Toate unitățile de execuție execută simultan aceeași instrucțiune asupra datelor din memoria proprie (există sisteme ce au și o memorie partajată pentru comunicații). Unitatea de control trebuie să permită tuturor elementelor de procesare să-și termine instrucțiunea curentă înainte de inițierea unei noi instrucțiuni, astfel că execuția instrucțiunilor trebuie sincronizată între toate unitățile de execuție. Ca și ordin de mărime numărul procesoarelor implicate într-o structură SIMD este de câteva mii.

Aplicabilitate: calculatoarele SIMD sunt folosite în cazul aplicațiilor paralele ce necesită un control fin

asupra datelor. Exemplu: rețele neuronale.

Exemple de implementări SIMD: ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP, Connection Machine CM-1, CM-2 (de la Thinking Machines Corporation), MassPar MP-1, MP-2.

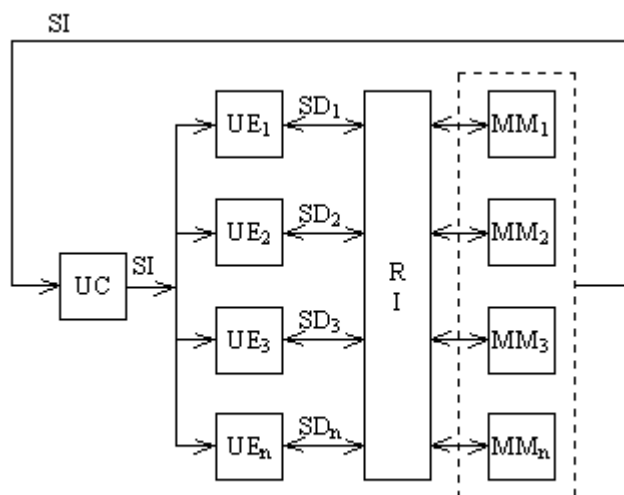


Figura 3.4 Arhitectura SIMD

Topologia rețelei de interconectare nu apare în clasificarea lui Flynn.

MISD (Multiple Instruction Single Data)

Arhitecturile MISD au mai multe elemente de procesare, fiecare executând un set diferit de instrucțiuni asupra unui singur flux de date. Acest lucru este realizabil în două moduri:

- același element din fluxul de date este prelucrat de toate procesoarele, fiecare executând propriile operații asupra respectivei date;
- un element din fluxul de date este prelucrat de primul procesor, rezultatul obținut este pasat mai departe celui de-al doilea procesor ș.a.m.d., formându-se astfel o macro-bandă de asamblare.

Singurul exemplu de implementare pentru acest tip de arhitecturi este C.mmp (calculator multimicroprocesor) construit la Carnegie-Mellon University. Acest calculator este reconfigurabil și poate opera în modurile SIMD, MISD și MIMD.

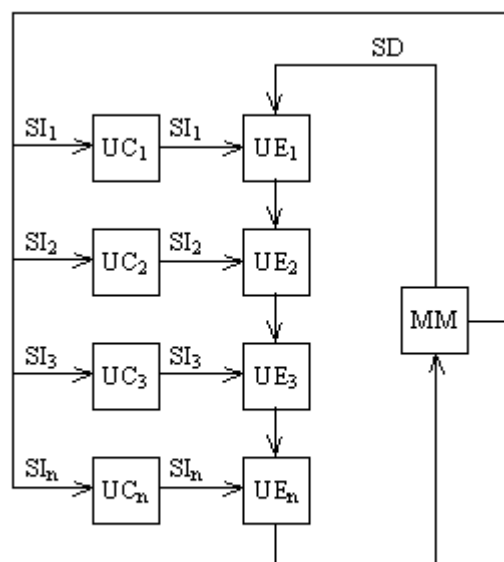


Figura 3.5 Arhitectura MISD

MIMD (Multiple Instruction Multiple Data)

Majoritatea sistemelor multiprocesor se pot încadra în această categorie. Un sistem de calcul MIMD are mai multe elemente de procesare interconectate, fiecare având propria unitate de control. Procesoarele lucrează fiecare asupra propriilor date executând asupra lor propriile instrucțiuni. Sistemele MIMD pot avea și memorie partajată. Operațiile executate de fiecare procesor sunt independente între ele, deci modul lor de operare este asincron.

Acest tip de arhitecturi sunt aplicabile în cazul aplicațiilor paralele (calcul paralel).

Exemple de implementare: C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, SGI/Cray Power

Challenge Array, SGI/Cray Origin-2000, HP/Convex SPP-2000, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, BBN Butterfly, Meiko Computing Surface (CS-1), FPS T/40000, iPSC.

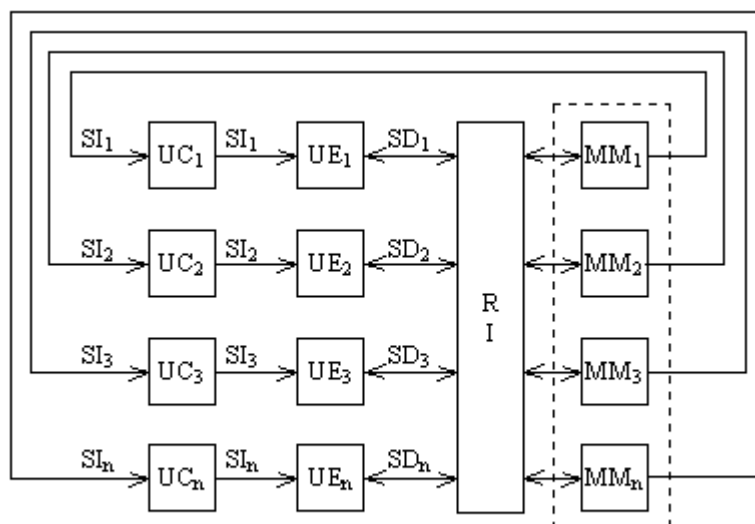


Figura 3.6 Arhitectura MIMD

O variantă între SIMD și MIMD sunt arhitecturile **SPMD** (Single Program Multiple Data), în care unitățile de procesare execută același segment de cod asupra unor date diferite, independent unul de celălalt (în mod asincron).

SIMD < SPMD < MIMD

Prezentare generală a microprocesoarelor Intel

Primele microprocesoare sunt produse la firma Intel în 1971: ele se "numeau" 4004 și 8008, pe patru și respectiv 8 biți. Primul microprocesor considerat "standard", care impune chiar o definiție a termenului și a unor concepte legate de această modalitate de prelucrare a informației este însă 8080 produs tot de firma Intel. Tot firma Intel este cea care lansează primul microprocesor care lucrează pe 16 biți - Intel 8086 (1978). În 1979, Intel face, aparent, un pas înapoi: lansează 8088 care este identic în interior cu 8086 dar în exterior lucrează pe 8 biți. Strategia firmei este limpede: mulți fabricanți nu sunt pregătiți să schimbe toate celelalte componente ale sistemelor de prelucrare pe 16 biți, așa că vor prefera încă microprocesoarele compatibile cu magistrala de 8 biți.

În tabelul de mai jos sunt prezentate generațiile și caracteristicile de bază ale microprocesoarelor Intel.

Tabelul 1 - Generațiile și caracteristicile de bază ale microprocesoarelor Intel

Tip/generație	Anul	Lățimea magistralei Date/Adrese, Biți	Cache Interior (L1), kB	Frecvența de tact a magistralei de memorie, (Mhz)	Frecvența de tact (interioară,) (Mhz)
8088/First	1979	8/20	None	4.77-8	4.77-8
8086/First	1978	16/20	None	4.77-8	4.77-8
80286/Second	1982	16/24	None	6-20	6-20
80386DX/Third	1985	32/32	None	16-33	16-33
80386SX/Third	1988	16/32	None	16-33	16-33
80486DX/Fourth	1989	32/32	8	25-50	25-50
80486SX/Fourth	1989	32/32	8	25-50	25-50
80486DX2/Fourth	1992	32/32	8	25-40	50-80
80486DX4/Fourth	1994	32/32	8+8	25-40	75-120
Pentium/Fifth	1993	64/32	8+8	60-66	60-200
Pentium MMX/Fifth	1997	64/32	16+16	66	166-233
Pentium Pro/Sixth	1995	64/36	8+8	66	150-200
Pentium II/Sixth	1997	64/36	16+16	66	233-300
Pentium	1999	64/36	32K+32K	100	650-1400

III/Sixth					
Pentium4/ Seventh	2000	64/36	64K+64K	100	1300-3800

Consacrarea definitivă a produselor Intel o face firma IBM care, în 1981, anunță primele calculatoare personale, IBM PC-XT, care folosesc 8088/8086. Aceste procesoare au introdus conceptul de segmentare a memoriei: memoria este împărțită în zone numite segmente de maxim 64 KB, iar cele patru registre de segment pot păstra adresele de bază ale segmentelor active. Aceste procesoare pot funcționa numai în *modul real*, care este un mod uniprocess, în care se execută un singur proces (program sau *task*) la un moment dat.

În 1982 Intel lansează microprocesorul 80286 (ignorăm că, între timp, multe firme ca National, Fairchild, RCA, Signetics etc. produc componente similare); acesta, deși tot pe 16 biți, introduce o serie de noi concepte fundamentale care tind să schimbe chiar noțiunea de microprocesor. Procesorul 80286 poate funcționa în *modul real* al procesoarelor precedente, dar dispune și de un mod de adresare virtual sau *mod protejat*. Acest mod utilizează conținutul registrelor de segment ca selectori sau pointeri în tabele ale descriptorilor de segment. Procesorul dispune de o unitate de gestiune a memoriei virtuale. În modul protejat, procesorul poate funcționa în regim multi-proces (*multitasking*), în care pot fi executate mai multe procese în mod concurrent. În acest mod se realizează o comutare prin hardware între procesele care se execută concurrent. Firma IBM lansează și ea o nouă generație de calculatoare personale: PC-AT ("Advanced Technology") care folosește Intel 80286.

Procesorul 80386 a introdus în cadrul arhitecturii *Intel* registre de 32 de biți, utilizate atât pentru păstrarea datelor, cât și pentru adresare. Pentru compatibilitate cu procesoarele anterioare, aceste registre s-au obținut prin extinderea registrelor de 16 biți, fiind posibilă utilizarea în continuare a vechilor registre, acestea constituind jumătatea de ordin inferior a registrelor de 32 de biți. A fost introdus un nou mod de funcționare, numit *mod virtual 8086*. Instrucțiunile originale au fost extinse cu noi forme care utilizează operanzi și adrese de 32 de biți, și au fost introduse instrucțiuni complet noi, ca de exemplu instrucțiuni pentru operații la nivel de bit.

Procesorul 80386 a introdus de asemenea mecanismul de paginare ca metodă de gestiune a memoriei virtuale. El a fost primul din cadrul familiei 80x86 care a utilizat o formă de prelucrare paralelă și o memorie încorporată *cache* cu informații despre până la 32 de pagini cel mai recent accesate.

Procesorul 80486 a fost primul din familia 80x86 la care unitatea de calcul în virgulă mobilă a fost integrată în același circuit cu unitatea centrală. Procesorului i s-a adăugat o memorie *cache* de nivel 1 (L1 – *Level 1*) de 8 KB. Au fost adăugați de asemenea noi pini și noi instrucțiuni care permit realizarea unor sisteme mai complexe: sisteme multiprocesor și sisteme care conțin o memorie *cache* de nivel 2 (L2 – *Level 2*).

Au fost dezvoltate versiuni ale procesorului 80486 în care au fost incluse facilități pentru reducerea consumului de putere, ca și alte facilități de gestiune a sistemului. Una din aceste facilități este noul mod de gestiune a sistemului (*System Management Mode* – SMM), pentru care s-a prevăzut un pin dedicat de întrerupere. Acest mod permite operații complexe de gestiune a sistemului (ca de exemplu gestiunea puterii consumate de diferitele subsisteme ale calculatorului), într-un mod transparent pentru sistemul de operare și pentru programele de aplicații. Facilitățile numite "*Stop Clock*" și "*Auto Halt Power down*" permit funcționarea unității centrale la o frecvență redusă a tactului pentru reducerea puterii consumate, sau chiar oprirea funcționării (cu memorarea stării).

În 1993, se lansează primul microprocesor al generației a cincea, numit din acest motiv *Pentium*. Procesorul *Pentium* a adăugat o nouă linie de execuție de tip *pipeline* a instrucțiunilor, pentru a se obține performanțe superscalare. Cele două linii de execuție a instrucțiunilor, numite U și V, permit execuția a două instrucțiuni pe durata unei perioade de tact. Capacitatea memoriei *cache* L1 a fost de asemenea dublat, fiind alocați 8 KB pentru instrucțiuni și 8 KB pentru date. Memoria *cache* pentru date utilizează protocolul MESI, care permite gestiunea memoriei *cache* atât prin metoda mai eficientă "*write-back*", cât și prin metoda "*write-through*" utilizată de procesorul 80486. Procesorul *Pentium* utilizează predicția salturilor pentru a crește performanțele construcțiilor care utilizează bucle de program. Registrele generale sunt tot de 32 de biți, dar s-au adăugat magistrale interne de date de 128 și 256 de biți pentru a crește viteza transferurilor interne, iar magistrala externă de date a fost extinsă la 64 de biți. Procesorului i s-a adăugat un controler avansat de întreruperi (*Advanced Programmable Interrupt Controller* – APIC) pentru a permite realizarea sistemelor cu mai multe procesoare *Pentium*, fiind adăugate de asemenea noi pini și un mod special de procesare dual pentru sistemele cu două procesoare.

Procesorul *Pentium Pro* este primul din cadrul familiei de procesoare P6. Acest procesor are o

arhitectură superscalară îmbunătățită, care permite execuția a trei instrucțiuni într-o stare (perioadă de tact, ceas). Procesorul *Pentium Pro*, ca și următoarele procesoare din familia P6, se caracterizează prin execuția dinamică a instrucțiunilor, care constă din analiza fluxului de date, execuția instrucțiunilor într-o altă ordine decât cea secvențială, o predicție îmbunătățită a salturilor și execuția speculativă. Pe lângă cele două memorii *cache* L1 de câte 8 KB, prezente și la procesorul *Pentium*, procesorul *Pentium Pro* dispune și de o memorie *cache* L2 de 256 KB, aflat în același circuit cu unitatea centrală, conectat cu aceasta printr-o magistrală dedicată de 64 de biți. Procesorul *Pentium Pro* are o magistrală de adrese extinsă la 36 de biți, astfel încât spațiul adreselor fizice este de până la 64 GB.

Procesorul *Pentium II* se bazează pe arhitectura *Pentium Pro*, la care s-au adăugat extensiile MMX (*Multimedia Extensions*). Memoria *cache* L2 a fost mutată în afara capsulei procesorului. Atât memoria *cache* L1 pentru date, cât și memoria *cache* L1 pentru instrucțiuni au fost extinse la 16 KB fiecare. Dimensiunea memoriei *cache* L2 poate fi de 256 KB, 512 KB, 1 MB sau 2 MB. Procesorul *Pentium II* utilizează diferite stări cu consum redus de putere, ca de exemplu "*AutoHALT*", "*Sleep*" și "*Deep Sleep*", pentru reducerea puterii consumate în perioadele de inactivitate.

Pentium III este ultimul din cadrul familiei P6, și se bazează pe arhitecturile procesoarelor *Pentium Pro* și *Pentium II*. Au fost adăugate 70 de noi instrucțiuni de tip SSE (*Streaming SIMD Extensions*) la setul de instrucțiuni existent. Acestea sunt destinate atât unităților funcționale existente la procesoarele precedente, cât și noii unități de calcul în virgulă mobilă de tip SIMD (*Single Instruction, Multiple Data*).

Primul din familia P7, numit *Pentium 4*, a primit și o nouă arhitectură cu o viteză de procesare mai performantă. În versiunea microprocesorului cu frecvența de tact de 3,06 Ghz a fost realizată o nouă tehnologie – *hyperthreading*. Această tehnologie permite ca procesele (programele) să fie divizate în două fluxuri de program pentru a fi procesate în paralel de microprocesor, ce mărește viteza de procesare. Pentru creșterea vitezei de prelucrare a datelor audio și video a fost introdus un set suplimentar din SSE instrucțiuni.

În noiembrie 2004 firma Intel a renunțat la producerea în serie a microprocesorului *Pentium 4* cu frecvența de 4 Ghz din cauza dificultăților apărute la răcirea lui.

Actualmente, firma Intel, încapsulează două și mai multe nuclee de microprocesor cu frecvențe reduse în circuitul unui microprocesor, ce exclude dificultățile cu răcirea lor.

Evoluția dispozitivelor pe care am descris-o până în acest moment se referă exclusiv la microprocesoarele ale căror caracteristici esențiale și arii de aplicații posibile s-au dezvoltat de la tipurile de bază lansate în anii 70. Noi vom denumi aceste dispozitive "microprocesoare" și le vom defini în acest capitol; vom detalia, în evoluția lor, conceptele esențiale care stau la baza funcționării lor în capitole următoare.

Există însă și alte direcții de dezvoltare a dispozitivelor de prelucrare a informației; un exemplu sunt așa numitele "procesoare cu set redus de instrucțiuni" (RISC) având ca reprezentanți procesoarele SPARC (produse de diferite firme), i860 (Intel), M88000 (Motorola) etc. De asemenea există procesoare specializate pentru anumite tipuri de prelucrări specifice cum sunt procesoarele digitale de semnal (DSP) și altele.

3.2 Microprocesoarele CISC/RISC

Multe microprocesoare au seturi de instrucțiuni ce includ mai mult de 100 – 200 instrucțiuni. Ele folosesc o varietate de tipuri de date și un mare număr de moduri de adresare. Tendința aceasta de a mări numărul de instrucțiuni a fost influențată de mai mulți factori, dintre care amintim:

- perfecționarea unor modele de procesoare existente anterior, pentru a pune la dispoziția utilizatorilor (programelor utilizator) cât mai multe funcții;
- adăugarea de instrucțiuni care să faciliteze translatarea din limbajele de nivel înalt în programe cod executabil (limbaj mașină);

Așa arhitecturi de microprocesoare au fost numite arhitecturi CISC - Complex Instruction Set Computer - calculator cu set complex de instrucțiuni. Câteva din caracteristici sunt:

- Multe instrucțiuni care prelucrează operanți din memorie;
- Format de *lungime variabilă* pentru instrucțiuni;
- Unitate de control microprogramată (micro-codată), avantajoasă din punctul de vedere al flexibilității implementării, dar lentă;
- Set complex (extins) de instrucțiuni și o mare varietate de moduri de adresare;
- Un număr relativ mic de registre în interiorul UCP.

- Utilizarea compilatoarelor optimizatoare - pentru a optimiza performanțele codului obiect;

Ideea simplificării setului de instrucțiuni, în scopul măririi performanțelor procesorului, provine din proiectele realizate la universitățile americane din Berkeley (RISC I, RISC II și SOAR) și Stanford (proiectul MIPS). Proiectele RISC (Reduced Instruction Set Computer - Calculator cu set redus de instrucțiuni) au urmărit ca instrucțiunile procesorului să fie de aceeași lungime, instrucțiunile să se execute într-o singură perioadă de ceas (cu ajutorul tehnicii de tip pipeline). La RISC se urmărește de asemenea ca accesările la memorie (consumatoare de timp) să se efectueze doar pentru operațiile de încărcare și stocare (arhitectura fiind numită în consecință: "load/store"), iar celelalte operații să se efectueze cu operanzi stocați în registrele interne ale UCP. Unele din proiectele de arhitecturi RISC folosesc un set mare de ferestre de registre pentru a accelera operațiile de apel al subrutinelor.

Rezumând, putem enumera câteva din elementele caracteristice pentru mașinile RISC:

- Acces la memorie limitat, doar prin instrucțiuni de *încărcare (load)* și *stocare (store)*;
- Format de *lungime fixă* pentru instrucțiuni, deci ușor de decodificat; caracteristică care contribuie la simplificarea structurii unității de control;
- structură *simplă a unității de control*, deci cu viteză mare de funcționare;
- Relativ *puține tipuri de instrucțiuni* (tipic sub 100 de instrucțiuni) și puține moduri de adresare (din nou această caracteristică contribuie și la simplificarea structurii unității de control);
- Tehnica de tip *pipeline* este utilizată și la arhitecturile CISC, dar la RISC tehnica este mai eficientă și mai ușor de implementat, datorită lungimii constante a instrucțiunilor;
- Un număr relativ mare de registre în interiorul UCP;

Așa cum s-a arătat mai sus, arhitecturile RISC restricționează numărul de instrucțiuni care accesează direct memoria principală. Cele mai multe instrucțiuni ale RISC presupun doar operații între registrele interne UCP. Pentru că instrucțiunile complexe nu există în setul de instrucțiuni, dacă este nevoie de ele, acestea se implementează prin rutine cu ajutorul instrucțiunilor existente. În final, într-un program executabil vor fi mai multe instrucțiuni decât la CISC, dar execuția pe ansamblu va fi mai rapidă. Formal, toate microprocesoarele x86 erau microprocesoare de tip CISC, dar microprocesoarele noi, începând de la Intel 486DX sunt microprocesoare CISC cu un nucleu RISC. Instrucțiunile microprocesoarelor x86 de tip CISC, înainte de executarea lor, sunt transformate într-un set simplu de instrucțiuni interne de tip RISC.

Multe microprocesoare moderne încorporează arhitecturi RISC, ca de exemplu ARM, DEC Alpha, SPARC, AVR, MIPS, POWER, PowerPC.

3.3 Banda de asamblare (pipeline)

Introducem noțiunea de *arhitectura suprascalară*. Pentru a explica această noțiune, urmărim utilizarea microinstrucțiunilor executate în paralel (tehnica "pipeline") folosite de microprocesoarele moderne. Începând cu microprocesorul i486 (fig. 3.7), Intel a introdus o bandă de asamblare, numită tehnica "pipeline", care constă din 5 segmente:

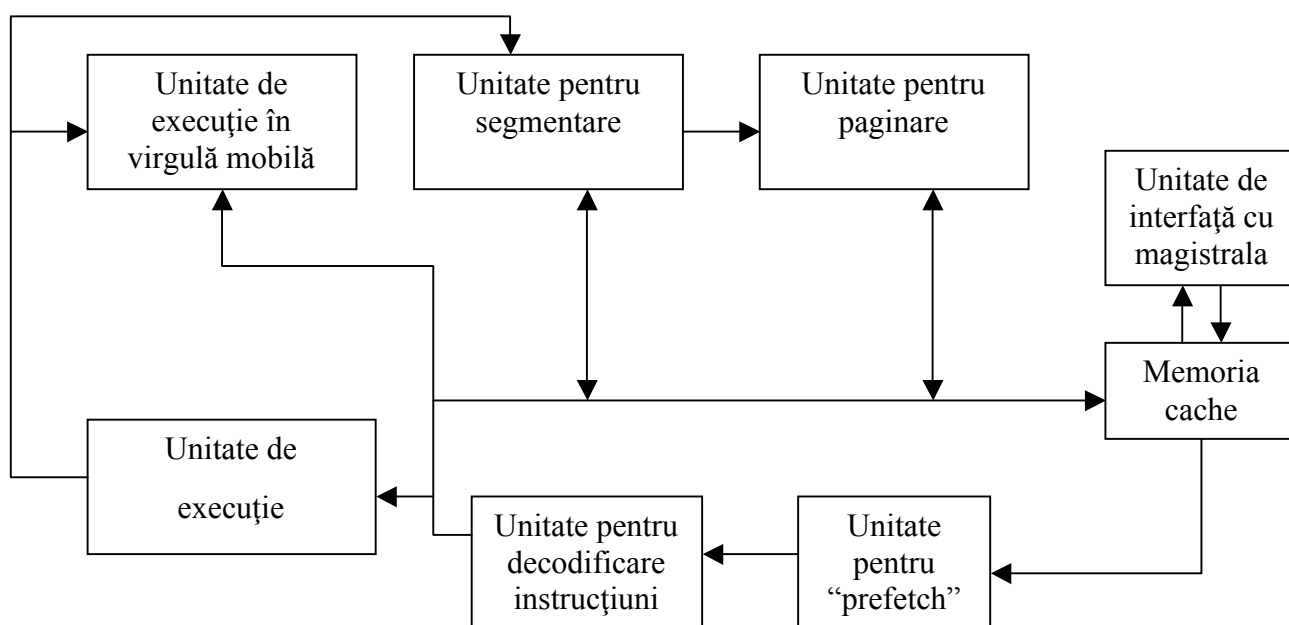


Figura 3.7

- Ciclul mașină „fetch”(Citirea instrucțiunii din cache sau din memoria internă);
- Decodificarea instrucțiunii – decodifică în microinstrucțiuni;
- Generarea adresei pentru localizarea operandilor în memorie;
- Prelucrarea instrucțiunii în UAL;
- Înscrierea rezultatului (unde va fi înscris rezultatul depinde de formatul instrucțiunii).

Toate aceste segmente sunt executate în paralel. Din unitatea „prefetch” instrucțiunea se transferă în unitatea pentru decodificarea instrucțiunii, și unitatea „prefetch” este liberă și poate citi următoarea instrucțiune. Deci, în interiorul microprocesorului se află 5 instrucțiuni în diferite segmente de execuție. Aceste segmente formează o Bandă de asamblare (pipeline).

Figura 3.8 ilustrează o bandă de asamblare cu 5 unități numite și stages (segmente, etape). Segmentul 1 extrage instrucțiunea din memorie și o plasează într-un registru tampon. Segmentul 2 o decodifică, determinându-i tipul și operandii. Segmentul 3 localizează și extrage operandii, fie din regiștri, fie din memorie. Segmentul 4 execută instrucțiunea, de obicei rulând operandii prin calea de date, iar segmentul 5 scrie rezultatul în regiștri.

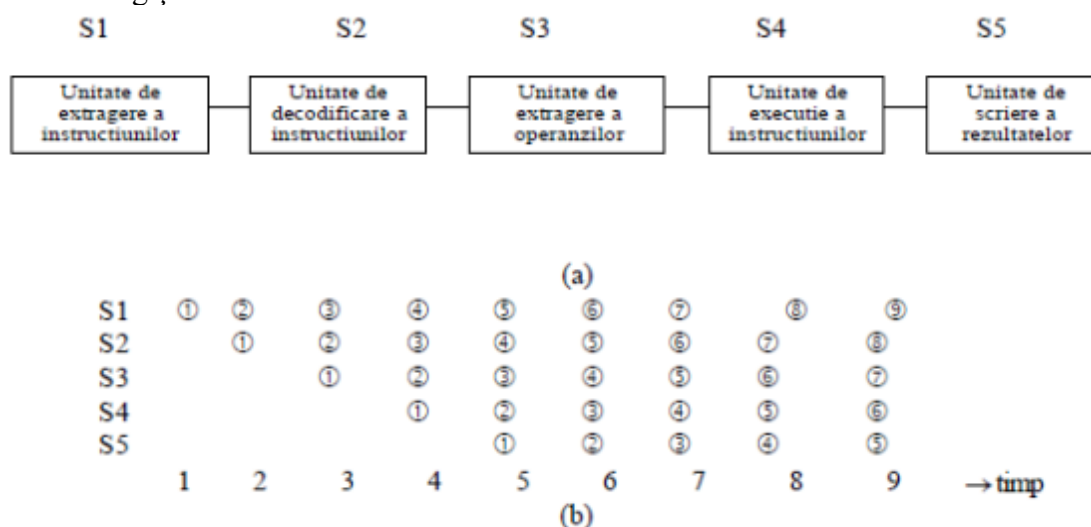


Figura 3.8 - O bandă de asamblare de 5 segmente (a)
Starea fiecărui segment în funcție de timp (b).

În figura 3.8 –b vedem cum operează o bandă de asamblare în funcție de timp. În ciclul 1, segmentul S1 lucrează asupra instrucțiunii 1 (o extragere din memorie). În ciclul 2, S2 decodifică instrucțiunea 1. Tot în ciclul 2, S1 extrage instrucțiunea 2. În ciclul 3, S3 extrage operandii pentru instrucțiunea 1, S2 decodifică instrucțiunea 2 și S1 extrage instrucțiunea 3.

Microprocesoarele ce includ o Bandă de asamblare se numesc microprocesoare cu *arhitectura scalară*, cele ce includ două și mai multe - microprocesoare cu *arhitectura suprascalară*. Microprocesorul Pentium include două Benzi de asamblare și poate executa 2 instrucțiuni pe durata unei perioade de ceas (clock, stare).

Date fiind avantajele benzii de asamblare, ar fi de dorit mai multe din acestea. În figura 3.9 este prezentată o posibilă proiectare a unui UCP în bandă de asamblare duală. Pentru a putea lucra în paralel, cele 2 instrucțiuni nu trebuie să-și dispute resursele (de exemplu registrele) și nici una nu trebuie să depindă de rezultatul celeilalte. Fie compilatorul trebuie să garanteze că ipoteza anterioară e respectată, fie conflictele sunt detectate și eliminate pe parcursul execuției, cu ajutorul unui hardware suplimentar.

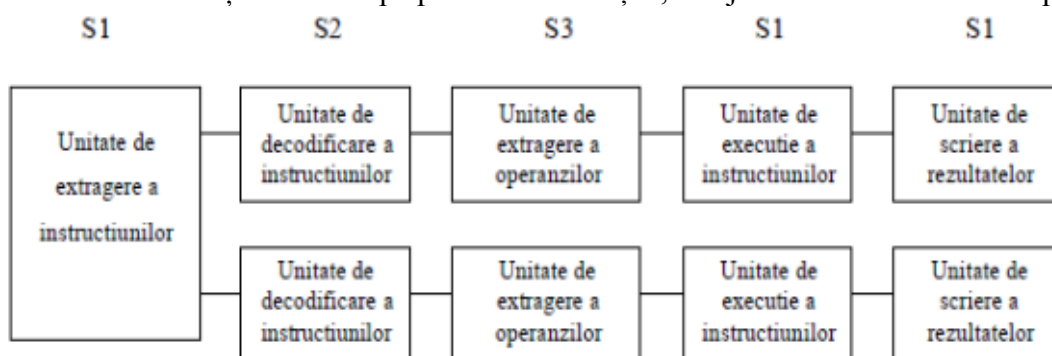


Figura 3.9 Banda de asamblare duală cu 5 segmente

Pentium are două benzi de asamblare asemănătoare cu cele din fig. 3.9, deși împărțirea între segmentele 2 și 3 (numite decode-1 și decode-2) este puțin diferită față de cea din exemplul nostru. Banda de asamblare principală, numită "u" pipeline, poate executa orice instrucțiune Pentium, în timp ce a doua bandă, numită "v" pipeline, poate executa doar instrucțiuni pentru întregi și o instrucțiune simplă în virgulă mobilă – FXCH. Reguli destul de complexe determină dacă instrucțiunile sunt compatibile, astfel încât să poată fi executate în paralel.

Alte UCP utilizează abordări cu totul diferite. Ideea de bază este de a avea o singură bandă de asamblare, dar cu mai multe unități funcționale, așa cum se observă în figura 3.10.

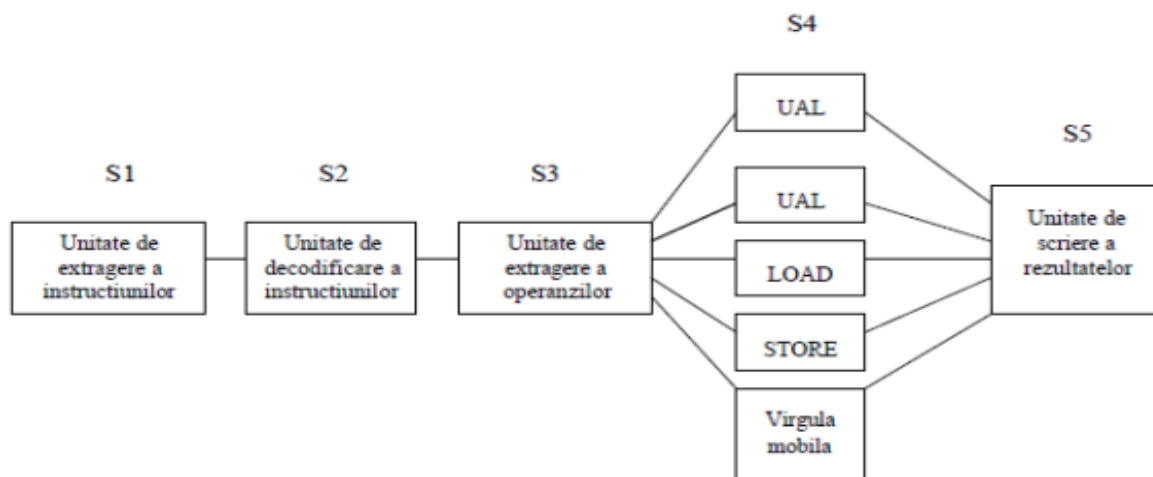


Figura 3.10 - Un procesor superscalar cu 5 unități funcționale

Procesorul Pentium II are o structură asemănătoare (sunt și diferențe) cu cea din fig. 3.10. Unitățile funcționale UAL din segmentul S4 pot executa instrucțiuni timp de un ciclu de ceas, iar cele care accesează memoria sau care lucrează în virgulă mobilă (mai lente), au nevoie de mai mult timp decât un ciclu de ceas pentru a-și executa funcția. După cum se poate observa, pot exista mai multe unități funcționale în segmentul S4.