Arhitectura Calculatoarelor

Introducere

Un calculator numeric este constituit dintr-un ansamblu de resurse fizice (hardware) şi de programe de sistem (software de sistem) care asigură prelucrarea automată a informaţiilor, în conformitate cu algoritmii specificaţi de utilizator prin programele de aplicaţii (software utilizator).

Arhitectura calculatorului cuprinde două componente principale:

a) arhitectura setului de instrucţiuni (ASI);

b) implementarea maşinii, cu cele două sub - componente:

- organizare;
- hardware.

Arhitectura setului de instrucţiuni este ceea ce trebuie să ştie un programator pentru a scrie programe în limbaj de asamblare, respectiv pentru a concepe şi 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 şi organizarea magistralelor, organizarea sistemului de memorie. Noţiunea de hardware (resurse fizice) e utilizată pentru a ne referi la aspectele specifice ale implementării calculatorului. Acestea includ proiectul logic de detaliu şi tehnologia de realizare a maşinii de calcul.

1. Reprezentarea funcțională a unui calculator

Un calculator poate fi descris atât sub aspect funcţional cât şi structural. Circuitele electronice ale unui calculator recunosc şi execută doar un set limitat de *instrucţiuni elementare*, codificate în formă binară. Aceste instrucţiuni sunt doar succesiuni de biţi (1 şi 0) pe care procesorul le înţelege - decodifică şi le execută. Indiferent de tipul de maşină, instrucţiunile 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 instrucţiunii următoare:

0000 0100 0000 0110

comandă adunarea conţinutului unui registru intern de 8 biţi (numit registrul al) cu valoarea imediată 6. Adesea, când utilizatorul este obligat să lucreze cu valori numerice binare, se foloseşte reprezentarea în hexazecimal, care este mai compactă şi mai uşor de citit. Codul, în hexazecimal, al instrucţiunii corespunzătoare succesiunii binare de mai sus este 04 06 hex. Şi acest mod de scriere a instrucţiunilor este însă complicat pentru programator. În assembler acestă instrucţiune se descrie **mov al,6.**

Scrierea instrucţiunilor 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 instrucţiuni, împreună cu regulile de organizare ale acestora într-un program. Totalitatea regulilor de scriere ale instrucţiunilor reprezintă *sintaxa* limbajului, iar totalitatea regulilor prin care se asociază o semnificaţie instrucţiunilor reprezintă *semantica* limbajului.

Calculatorul poate executa numai instrucţiuni exprimate intern sub forma unor şiruri de cifre binare. Programele în care instrucţiunile sunt scrise sub această formă se numesc programe în *limbaj maşină*. Limbajul maşină este caracteristic fiecărui tip de calculator. Scrierea şi introducerea programelor sub această formă este dificilă. Pentru simplificarea scrierii programelor, au fost create limbaje în care fiecărui cod de instrucţiune i s-a ataşat un *nume mnemonic*. Acest mod de reprezentare a instrucţiunilor maşină se numeşte *limbaj simbolic* sau *limbaj de asamblare*. Acest limbaj permite utilizatorului să realizeze codificări simbolice ale instrucţiunilor şi ale adreselor de memorie.

Pentru a se executa un program în limbaj de asamblare, acesta trebuie translatat în prealabil în limbaj maşină, printr-un proces numit *asamblare*. Programul care se translatează se numeşte *program sursă*, iar cel rezultat în urma translatării se numeşte *program obiect*. Operaţia de translatare a

programului sursă în program obiect este executată de un program special numit *asamblor*.
O instrucţiune în limbaj de asamblare corespunde unei instrucţiuni în limbaj maşină, deosebirea dintre ele constând numai în modul de reprezentare. Deci, fiecare instrucţiune în limbaj de asamblare este translatată într-o singură instrucţiune în limbaj maşină.

Un limbaj de asamblare este specific unui calculator, astfel încât trebuie să se cunoască instrucţiunile şi 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 instrucţiuni într-un limbaj de nivel înalt îi corespunde o succesiune de instrucţiuni în limbaj maşină. Translatarea în limbajul maşină se poate realiza cu ajutorul unui *compilator*, 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 instrucţiune în limbajul de nivel înalt într-o succesiune de instrucţiuni maşină, acestea fiind executate imediat. În acest caz nu se generează un program executabil. Viteza de execuţie este însă redusă, deoarece fiecare instrucţiune trebuie interpretată chiar dacă ea este executată în mod repetat.

Uneori este mai convenabil să se considere că există un calculator ipotetic sau o *maşină virtuală*, a cărui limbaj maşină este un anumit limbaj de nivel înalt. Un asemenea calculator ar executa direct instrucţiunile limbajului de nivel înalt, fără a fi necesară utilizarea unui translator (compilator) sau interpretor. Chiar dacă implementarea unei maşini virtuale care să lucreze direct cu un limbaj de nivel înalt ar fi prea costisitoare, se pot scrie programe pentru această maşină, 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 maşini virtuale pe mai multe nivele (Figura 1.1).

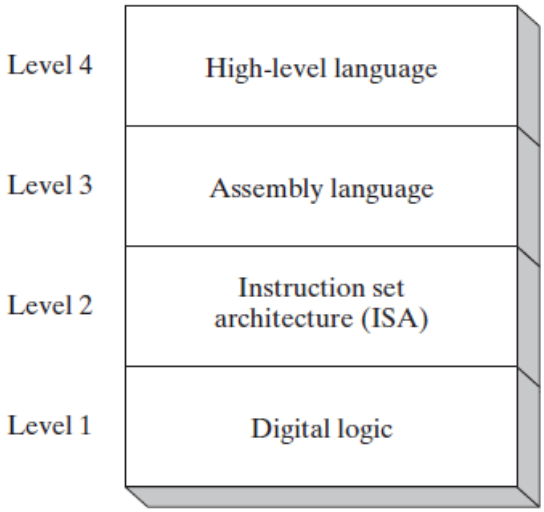


Figura 1.1 - Ierarhia de nivele a unui calculator modern

Nivelul 1, numit *nivelul logicii digitale*, este reprezentat de componentele hardware ale calculatorului (maşina fizică). Circuitele acestui nivel execută instrucţiunile maşină ale nivelului 2. Elementele de bază ale acestor circuite sunt *porţile 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), şi are ca ieşire o funcţie simplă a acestor intrări, de exemplu ŞI logic, SAU logic.

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

De menţionat că la anumite calculatoare nivelul de microprogram lipseşte. La aceste calculatoare, instrucţiunile maşinii convenţionale 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 aplicaţie, fiind numit *nivelul limbajelor de nivel înalt*. Programele scrise în aceste limbaje sunt translatate în o mulţime de instrucţiuni a limbajului nivelului 3 cu ajutorul compilatoarelor sau interpretoarelor.

Fiecare nivel reprezintă o abstractizare distinctă, cu diferite obiecte şi operaţii. Setul tipurilor de date, a operaţiilor şi facilităţilor 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 părţi 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 noţiuni pregătitoare.

Unitatea de bază a informaţiei memorate în calculator este **bitul.** Un bit reprezintă o cifră binară (de aici şi 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 şi 1, capabil să memoreze un bit de informaţie.

Un grup de bistabili formează un **registru.** De exemplu, 8 bistabili formează un registru de 8 biţi. Informaţia care se poate memora într-un asemenea registru poate fi codificată în binar, de la valoarea 00000000 (toţi biţii egali cu 0), până la valoarea 11111111 (toţi biţii egali cu 1). Este uşor de văzut că numărul combinaţiilor care pot fi memorate este 256 (2 la puterea a 8-a). În general, un registru de n biţi va putea memora 2^n combinaţii distincte. Aceste combinaţii se numesc **octeţi** sau **bytes** (dacă n = 8), respectiv **cuvinte** (dacă n = 16, 32 etc).

2.2 Registrele microprocesoarelor x86-64

Fiecare program la execuţie obţine anumite resurse ale microprocesorului. Aceste resurse (registre) sunt necesare pentru executarea şi păstrarea în memorie a instrucţiunilor şi datelor programului, a informaţiei despre starea curentă a programului şi a microprocesorului.

Microprocesoarele pe 32 biţi funcţionează în diferite moduri, ce determină mecanismele de protecţie şi de adresare a memoriei: modul real 8086 (pe 16 biţi), modul virtual 8086 (V8086), modul protejat pe 32 biţi (inclusiv protejat pe 16 biţi). Modul de funcţionare a microprocesorului este impus de sistemul de operare (SO) în conformitate cu modul definit de aplicaţii (task-uri).

În microprocesoarele pe 64 biţi au fost introduse noi moduri de funcţionare:

- Modul pe 64 biţi (64-bit mode) – acest mod susţine adresarea virtuală pe 64 biţi şi extensiile registrelor pe 64 biţi. În acest mod este folosit numai *modelul plat de memorie* (un segment comun pentru cod, date şi stivă).

- Modul de compatibilitate (compatibility mode) permite SO să execute aplicaţii pe 32 şi 16 biţi. Pentru aplicaţii microprocesorul reprezintă un microprocesor pe 32 biţi cu toate atributele modului protejat, cu mecanismele de segmentare şi paginare.

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

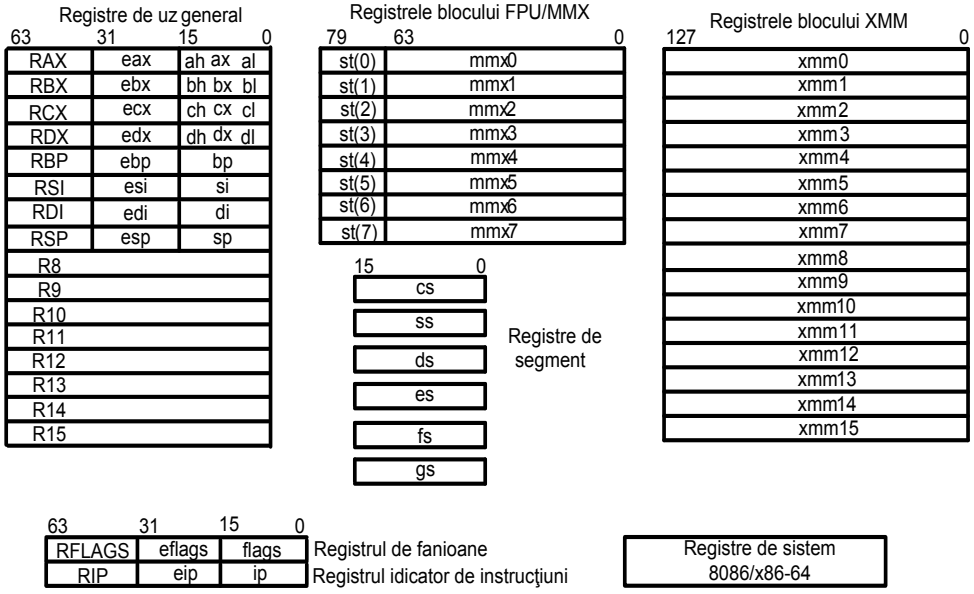


Figura 2.1 Registrele microprocesoarelor x86-64

Structura şi destinaţia regiştrilor setului FPU/MMX, XMM, de sistem vor fi prezentate în capitolul „Modelul program ale microprocesoarelor”.

2.2.1 Regiştrii de uz general

În figura 2.2 sunt prezentaţi regiştrii de uz general pe 64 biţi.

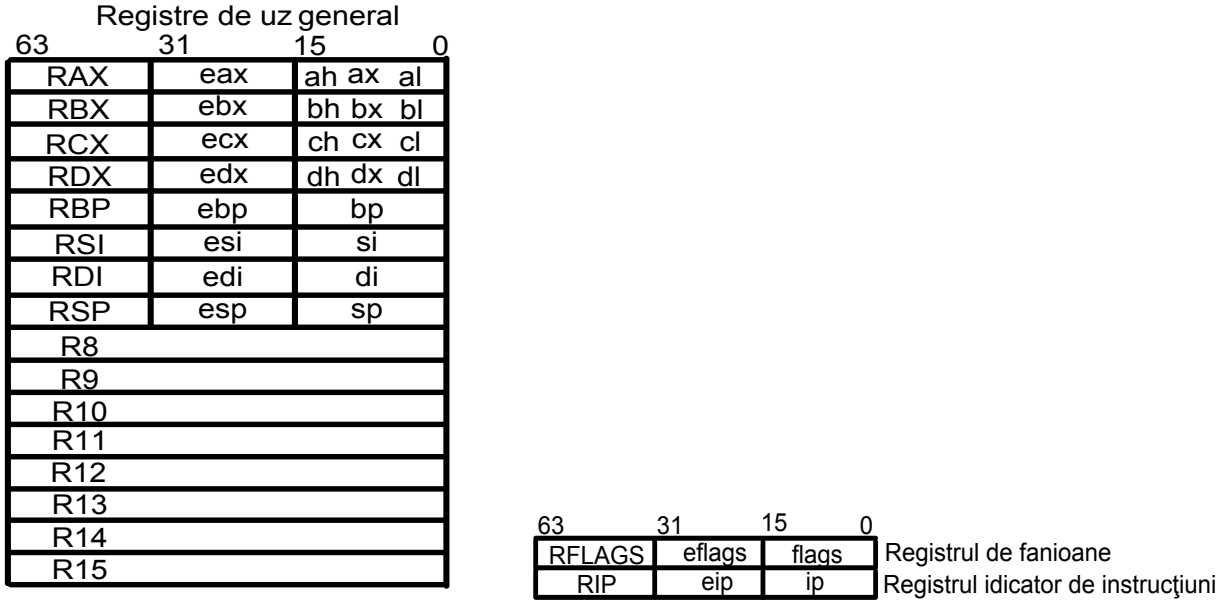


Figura 2.2 - Regiştrii de uz general pe 64 biţi

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

2.3 Modul de compatibilitate pe 16 biţi

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

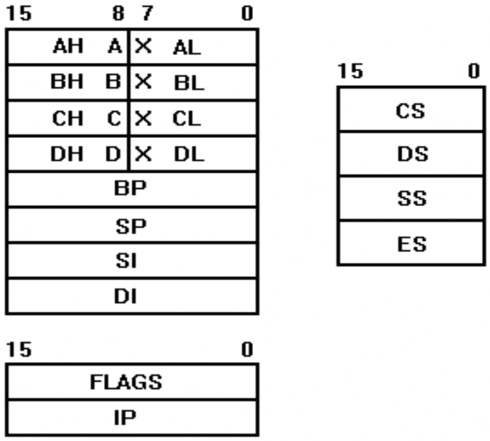


Figura 2.3

Toate registrele sunt de 16 biţi. O serie de registre (AX, BX, CX, DX) sunt disponibile şi la nivel de octet, părţile mai semnificative fiind AH, BH, CH şi DH, iar cele mai puţin semnificative, AL, BL, CL şi 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 destinaţie

Registrul notat FLAGS cuprinde flagurile (biţi indicatori) procesorului, sau bistabililor de condiţie, iar registrul IP (instruction pointer) este registrul de instrucţiuni.

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 registrul CX este folosit ca contor în instrucţiunea LOOP de ciclare, care îl modifică implicit şi-i testează valoarea, registrul DX este folosit în instrucţiunile de împărţire si înmulţire, iar registrul BX este folosit pentru adresarea datelor (operanzilor).

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 instrucţiuni, o zonă specială pentru stivă şi o zonă suplimentară pentru date. Fiecare din aceste zone pot avea până la 64K octeţi şi poartă denumirea de segment. Astfel există segmentul de date (Data Segment), segmentul de instrucţiuni, cod (Code Segment), segmentul de stivă (Stack Segment) si segmentul suplimentar de date (Extra Segment).

Este posibil ca cele 4 segmente sa fie suprapuse total sau parţial. 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 instrucţiuni care se va executa, iar perechea (SS:SP) indică totdeauna adresa vârfului stivei. Registrele DS şi ES conţin adresele segmentelor de date şi sunt folosite pentru a accesa date.

Dacă segmentul de date începe de la locaţia de memorie 1234h atunci DS va conţine valoarea 1234h. Există instrucţiuni pentru încărcarea adreselor de memorie în registrele segment.

Registrele pointer (SP si BP) se folosesc pentru calculul offsetului (distanţei faţă de începutul unui segment) din cadrul segmentului. Cele două registre pointer sunt: pointerul de stiva SP (Stack Pointer) si pointerul de baza (Base Pointer). SP si BP sunt registre de 16 biti.

Registrul SP reţine adresa efectivă (offsetul) a vârfului stivei (figura 2.4). Adresa fizică a vârfului stivei SS:SP este dată de perechea de registre SS si SP, registrul SS conţine adresa de început al segmentului de stiva iar SP conţine offsetul din acest registru (adică distanţa in octeţi de la începutul registrului de stiva):

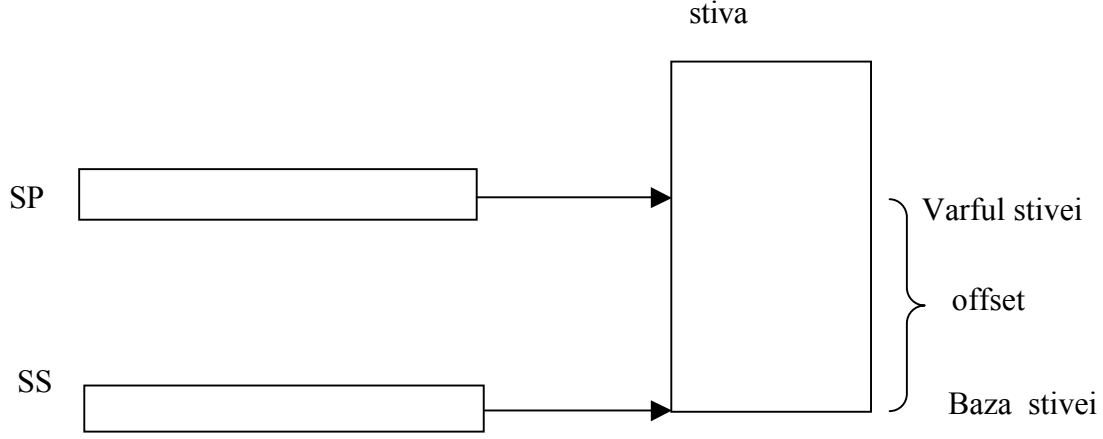


Figura 2.4

Registrul 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 biti sunt: SI (Source Index) si DI (Destination Index). Registrele de index sunt folosite pentru accesul la elementele unui tablou sau a unei tabele. Aceste registre sunt folosite îndeosebi in prelucrarea şirurilor de caractere.

Registrul de instrucţiuni (Instruction Pointer) conţine 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 şi incrementează corespunzător valoarea lui IP, cu numărul de octeți ai codului instrucțiunii ce va fi executată. Uneori acest registru se numeşte numărător de program.

Registrul de flag-uri (fanioane) (bistabili de condiţie) al modului 8086 are configuraţia din figura 2.5. O serie de flag-uri sunt flag-uri de stare: acestea sunt poziţionate la 0 sau la 1 ca urmare a unor operaţii aritmetice sau logice, conţin informaţii despre ultima instrucțiune executată. Celelalte flag-uri controlează anumite operaţii 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.

Semnificaţia flag-urilor (biţilor) 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 operaţii de adunare sau de scădere.

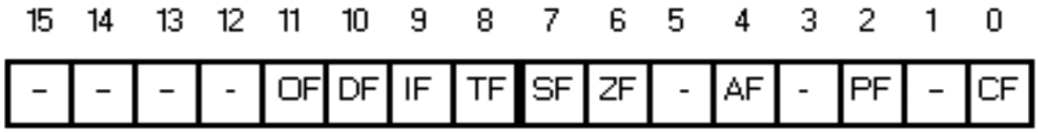


Figura 2.5 - Registrul de flag-uri al procesorului 8086

– PF (Parity Flag, flag de paritate) - este poziţionat în aşa fel încât numărul de biţi egali cu 1 din octetul cel mai puţin semnificativ al rezultatului, împreună cu flag-ul PF, să fie impar; altfel formulat, suma modulo 2 (XOR) a tuturor biţilor din octetul c.m.p.s. şi a lui PF să fie 1.

– AF (Auxiliarry 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 poziţionat la 1 dacă rezultatul operaţiei este 0.

– SF (Sign Flag, bistabil de semn) - este poziţionat la 1 dacă b.c.m.s. al rezultatului (bitul de semn) este 1.

– OF (Overflow Flag, bistabil de depăşire) - este poziţionat la 1 dacă operaţia a condus la o depăşire de domeniu a rezultatului (la operaţii cu sau fără semn).

– TF (Trap Flag, bistabil de urmărire) - dacă este poziţionat la 1, se forţează o întrerupere, pe un nivel predefinit, la execuţia fiecărei instrucţiuni; 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 poziţionat la 1, procesorul ia în consideraţie întreruperile hardware externe; altfel, acestea sunt ignorate.

– DF (Direction Flag, bistabil de direcţie) - precizează sensul (crescător sau descrescător) de variaţie a adreselor la operaţiile cu şiruri de octeţi sau de cuvinte.

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

2.3.1 Formarea adresei fizice

Procesorul 8086 dispune de adrese pe 20 de biţi, fiind capabil să adreseze 1 megaoctet de memorie (2^{20}). Se pune problema cum se formează adresa fizică pe 20 de biţi (deci pe 5 cifre hexa), deoarece toate registrele procesorului sunt de 16 biţi, putând codifica adrese în domeniul 0000...0FFFFH (pe 4 cifre hexa), deci într-un spaţiu 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 şi adresa de segment se poate reprezenta tot pe 16 biţi. 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 depăşi 64 KO, deplasamentul se poate memora tot pe 16 biţi. Deplasamentul poate fi o constantă sau conținutul unui registru care permite adresarea memoriei.

În concluzie, pentru adresarea unui octet de memorie, se folosesc două entităţi pe 16 biţi: o adresă de segment (conţinută obligatoriu într-un registru de segment) şi un deplasament. Deoarece ambele entităţi sunt pe 16 biţi, se vorbeşte de adrese (sau pointeri) de 32 de biţi, deşi adresa fizică este doar pe 20 de biţi.

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

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

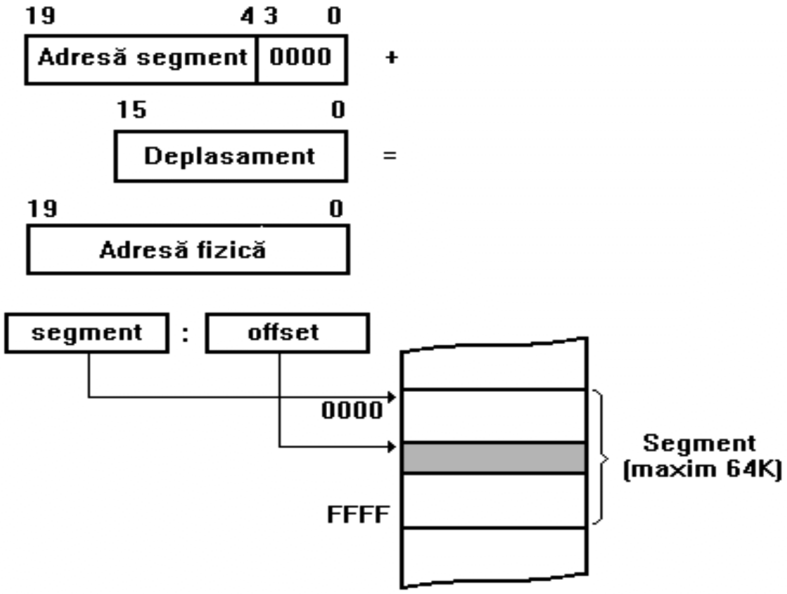


Figura 2.6 - Formarea adresei fizice

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

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

2.3.2 Definirea segmentelor. Structura programelor

Segmentele logice conţin cele trei componente ale unui program: cod, date si 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** (sfârşit logic al programului)

În principiu, segmentul de cod va cuprinde programul executabil (instrucţiuni), 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 sfârşitul liniei curente, este considerat comentariu. Structura programului este:

```
.MODEL small
.STACK 512
.DATA

definirea datelor
```

```
.CODE

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 aloca un segment de lungime 512 octeți). Directiva rezervă (nu şi inițializează) zona dedicată stivei. Acțiunea de inițializare este opțională. Putem scrie si .stack 512.

Directiva **.code** precede segmentul de program. Încărcarea acestui segment este realizata automat de către DOS. In 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 şi 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 asigurata tot de către componentele sistemului de operare.

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

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

- **tiny** - toate segmentele (date, cod, stivă) se pot genera într-un spaţiu de 64KO şi formează un singur grup de segmente. Se foloseşte la programele de tip COM.
- **small** - datele şi stiva sunt grupate într-un singur segment iar codul în alt segment. Fiecare din acestea nu depăşesc 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 sfârşitul logic al unui modul de program şi e obligatorie în toate modulele. Tot ce se găseşte în fişierul 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 biţi (microprocesor de 32 biţi)

Pentru modul de compatibilitate pe 32 biți, sunt utilizate numai părțile inferioare pe 32 biți ai registrelor microprocesoarelor pe 64 de biți şi 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 biţi a fost adăugat un set de instrucțiuni.

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

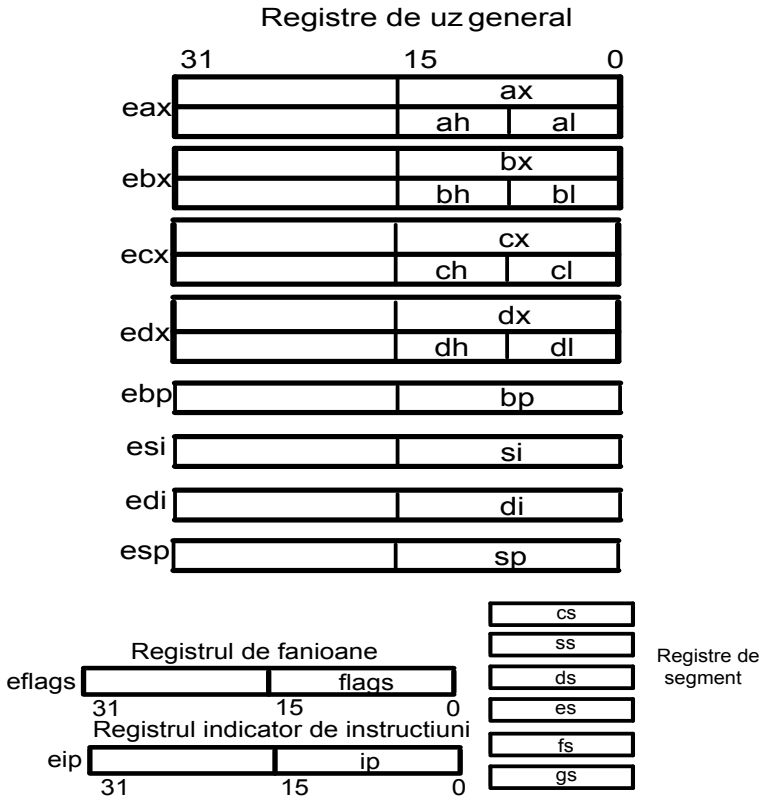


Figura 2.7 - Setul de registre de uz general

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

Analog registrele IP si FLAGS de 16 biţi sunt extinse la 32 biţi in cazul registrelor EIP şi ELAGS de 32 biţi. Registrul FLAGS se regăseşte în primii 16 biţi ai registrului EFLAGS.

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

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

Semnificaţia registrelor segment în cazul microprocesoarelor de 32 biţi 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 şi drepturile de acces asociate acestuia. Astfel adresa de bază poate fi specificată pe 32 biţi 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 informaţii este bitul, în care se poate memora o cifra binara, 0 sau 1.

De obicei informaţia de prelucrat se reprezintă pe segmente contigue de biţi denumite tetrade, octeţi, cuvinte, dublu cuvinte, quadwords si tenbytes.

Tetrada. Tetrada este o secvenţă de 4 biţi, numerotaţi 0,1,2,3 de la dreapta la stânga, bitul 0 fiind cel mai puţin 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 secvenţa de 8 biţi. Octetul este unul dintre cele mai importante elemente (celule) de memorare adresabile. Cei 8 biţi ai unui octet sunt numerotaţi 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) conţine biţii 0, 1, 2, 3, iar cea superioara (din stânga) conţine biţii 4, 5, 6, 7 ai octetului.

Cuvântul(Word). Cuvântul este o secvență de 2 octeţi, respectiv 16 biţi, numerotaţi de la dreapta spre stânga, astfel 0, 1, 2 .....14, 15. Bitul cel mai semnificativ este bitul 15. Primul octet(inferior) din cuvânt conţine biţii 0, 1, 2, 3, 4, 5, 6, 7, iar al doilea octet(superior), biţii 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 biţi sau în doi octeţi de memorie. In memorie, octetul inferior (biţii 0-7) este memorat la adresa mai mică, iar octetul superior (biţii 8-15) la adresa cea mai mare.

De exemplu cuvântul 4567h se reprezintă intr-un registru de 16 biţi 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 octeţi, 32 biţi), reprezintă un dublu cuvânt. Cei 32 de biţi 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. In memorie, octetul 1-cel mai puțin semnificativ este memorat la adresa cea mai mica, 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 octeţi succesivi de memorie). Cei 64 biţi ai unui qword sunt numerotați de la dreapta la stânga astfel: 0, 1, 2, ...62, 63. Bitul cel mai semnificativ este bitul 63. In memorie octetul 1 se reprezintă la adresa cea mai mica, 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. In memorie octetul cel mai puțin semnificativ (biții 0-7) se reprezintă la adresa cea mai mica, 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 (pseudoinstructiuni) specifice pentru definirea lor.

a) Byte (octet).

Acest tip de date ocupa 8 biţi, adică un octet (byte). Informaţia dintr-un octet poate fi: un *întreg fără semn* cuprins intre 0 si 225, *un întreg cu semn* cuprins intre –128 si 127, sau un *caracter* ASCII.

Definirea datelor de tip byte se face cu ajutorul directivelor BYTE şi 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 şi 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 simbolica *alfa* (offset, etichetă de date), se va genera secvenţa de octeți, reprezentata in hexazecimal :

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

Valoarea binara 11011b va fi generata la adresa alfa+3.

Definirea şirurilor 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, nedefiniţi
BYTE    4   DUP("STACK") ; 20 bytes: "STACKSTACKSTACKSTACK"
```

b) WORD (cuvânt).

Un cuvânt ocupa doi octeți (16 biți) si poate fi reprezentat intr-un registru de 16 biți sau in 2 octeți consecutivi de memorie. Numerotarea biților in 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:

Informaţia memorata intr-un cuvânt poate fi :

-un întreg pe 16 biți cu semn (bitul 15 este bitul de semn), cuprins intre -2^{15} si $2^{15}-1$,

- un întreg pe 16 biți fără semn, cuprins intre 0 si 2^{16}

- o adresa de memorie de 16 biți.

Reprezentarea celor 2 octeți ai cuvântului in memorie se face astfel încât octetul cel mai puțin semnificativ este memorat la adresa cea mai mica. De exemplu: daca valoarea 2345h este memorata 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 şi SWORD:

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

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

Fie secvenţa de directive:

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

In 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 octala 2476o este generată de la adresa beta +6.

c**) Double WORD**(dublu cuvânt)

Un dublu cuvânt ocupa 2 cuvinte sau 4 octeți (32 biți) si poate fi reprezentat in 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 memorata într-un dublu cuvânt poate fi:

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

Generarea datelor de tip dublu cuvânt se poate face folosind directivele DWORD şi 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 şi cu directiva DD (*Define Double Word*):

Reprezentarea celor doua cuvinte a unui dublu cuvânt de memorie se face astfel încât cuvântul cel mai puțin semnificativ este memorat la adresa cea mai mica. 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.

Secvenţa de directive :

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

d) QUAD – WORD (8 octeți)

Tipul Quad – word (QWORD) ocupa 8 octeți şi este reprezentat in 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 biţi.

Informaţia stocata intr-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 şi cu directiva DQ (*Define Quad – word*):

```
quad1 DQ 1234567812345678h
```

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

e) Ten Bytes

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

Informaţia stocata intr-un tbyte poate fi: un număr întreg reprezentat ca o secvenţa de cifre BCD (format împachetat) cu sau fără semn, sau un număr real in precizie extinsa.

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 şi 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 – defineşte variabile în virgulă mobilă, în simpla precizie pe 32 biţi;
- REAL8 – defineşte variabile în virgulă mobilă, în dubla precizie pe 64 biţi;
- REAL10 – defineşte variabile în virgulă mobilă, cu precizie extinsă pe 80 biţi.

```
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 şi cu directivele:

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

Gama valorilor definite pot fi:

REAL4 - 1.18×10^{-38} până 3.40×10^{38}
REAL8 - 2.23×10^{-308} până 1.79×10^{308}
REAL10 - 3.37×10^{-4932} până 1.18×10^{4932}

2.7 Setul de instrucţiuni MASM

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

Setul de instrucţiuni este grupat în 6 clase:

- **instrucţiuni de transfer**, care deplasează date între memorie sau porturi de intrare/ieşire şi registrele microprocesorului, fără a executa nici un fel de prelucrare a datelor;
- **instrucţiuni aritmetice şi logice**, care prelucrează date în format numeric;
- **instrucţiuni pentru şiruri**, specifice operaţiilor cu date alfanumerice;
- **instrucţiuni pentru controlul programului**, care în esenţă se reduc la salturi şl la apeluri de proceduri;
- **instrucţiuni** specifice **întreruperilor hard şi soft**;

– **instrucţiuni pentru controlul procesorului**.

Această împărţire este realizată după criterii funcţionale. De exemplu, instrucţiunile PUSH şi POP sunt considerate ca instrucţiuni de transfer, deşi, la prima vedere, ar putea fi considerate instrucţiuni specifice procedurilor. Acelaşi lucru despre instrucţiunile IN şi OUT, care interfaţează microprocesorul cu lumea exterioară: ele sunt considerate instrucţiuni de transfer, deşi ar putea fi considerate instrucţiuni de intrare/ieşire. Intrările şi ieşirile sunt însă cazuri particulare de transfer.

Fiecare categorie de instrucţiuni este însoţită de specificarea explicită a flag-urilor (indicatorilor de condiţie) care sunt modificaţi în urma execuţiei.

Structura generală a instrucţiunilor x86 este următoarea:

[eticheta:] mnemonic [operanzi][; comentariu]

Instrucţiunile pot conţine zero, unu, doi sau trei operanzi. Omitem câmpurile etichetă şi comentariu:

mnemonic
mnemonic [*destinatie*]
mnemonic [*destinatie*],[*sursa*]
mnemonic [*destinatie*],[*sursa-1*],[*sursa-2*]

Sunt trei tipuri de bază de operanzi:

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

2.7.1 Instrucţiuni pentru transferuri de date, instrucţiuni în aritmetica binara şi în aritmetica BCD. Noţiuni teoretice

Instrucţiunile de transfer permit copierea unui octet sau cuvânt de la sursa la destinaţie. Destinaţia poate fi un registru, locaţie de memorie sau un port de ieşire, iar sursa poate fi un registru, o locaţie de memorie, constante sau port de intrare. Ca regula generală destinaţia şi sursa nu pot fi ambele locaţii de memorie. În specificarea sursei şi destinaţiei se vor folosi notaţiile:

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

Instrucţiuni de transfer

a) Instrucţiunea MOV (Move Data).

Forma generală a instrucţiunii Mov este:

mov dest, sursa ; [dest] ← [sursa]

realizează transferul informaţiei de la adresa efectiva data de *sursa* la *dest.*

Restricţii:

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

Structura instrucţiunii **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
```

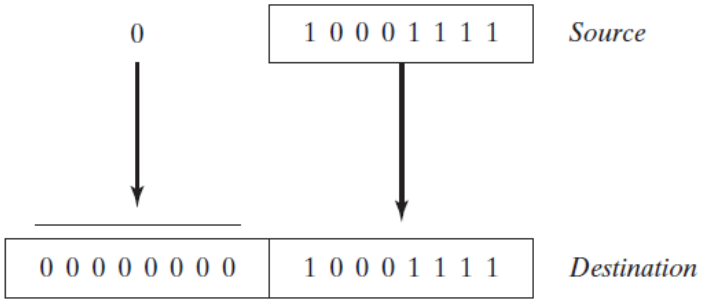
```
.data
alfa dw  1234h
beta db 56h
.code
mov  ax, alfa;          transfera conţinutul adresei alfa în ax
mov  bx, offset beta;   transfera adresa efectiva alfa în bx
mov  al, 75h;           transfera 75h în al
mov  cx, [100];         transfera conţinutul adresei 100 în cx
mov  [di], bx;  transfera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

Instrucţiunea **MOVZX** (*move with zero-extend*)
Copie conţinutul sursei în destinaţie cu extinderea valorii întroducând zerouri. Această instrucţiune este utilizată numai petru 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
```

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

Copie conţinutul sursei în destinaţie cu extinderea valorii întroducând unităţi. Această instrucţiune este utilizată numai petru 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 = FFFFFF9Bh
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 = FFFFFF9Bh
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) Instrucţiunea XCHG (Exchange Data)

Interschimbă sursa cu destinaţia. Forma generală:

XCHG dest, sursa

Restricţii:

- registrele de segment nu pot apărea ca operanzi;
- cel puţin un operand trebuie sa 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 conţinutului a doi operanzi din memorie op1 şi op2 se poate face prin secvenţa de instrucţiuni:

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

c) Instrucţiunea XLAT (Translate)

Forma generală:

XLAT

Instrucţiunea nu are operanzi, semnificaţia fiind:

[al]← ds: [[bx]+[al]]

adică se transfera în *al* conţinutul octetului de la adresa efectiva : *[bx]+[al]*.

Instrucţiunea se foloseşte la conversia unor tipuri de date, folosind tabele de conversie, adresa acestor tabele se introduce în *bx*, iar în *al* se introduce poziţia elementului din tabel. De exemplu: conversia unei valori numerice cuprinsă intre 0 şi 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 hexazecimala b.

c) Instrucţiunile IN (Input Data) şi OUT (Output Data)

Instrucţiunea **IN** execută o citire de 8, 16, 32 biţi de la portul de intrare. Şi invers, instrucţiunea **OUT** execută o scriere de 8, 16, 32 biţi într-un port. Sintaxa este urmatoarea:

```
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-biţi, AX pentru transferuri pe 16-biţi şi EAX pentru transferuri pe 32-biţi.

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) Instrucţiunea LEA (Load Effective Address)

Are ca efect încarcărea adresei efective (offsetul) intr-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]
```

Acelaşi efect se obţine folosind operandul OFFSET în Instrucţiunea MOV:

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

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

Forma generală:

LDS reg, sursa

unde:

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

Are ca efect transferul unei adrese complete în perechea de registre *ds* şi *reg* specificat în instrucţiune, 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) Instrucţiunea LAHF (Load AH with FLAGS)

Instrucţiunea încarcă în registrul AH octetul dat de partea cea mai puţin semnificativa a registrului FLAGS, ce conţine indicatorii. Instrucţiunea nu are operanzi.

AH ← FLAGS $_{0\text{-}7}$

Ex.

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

j) Instrucţiunea SAHF (Store AH into FLAGS)

Instrucţiunea încarcă în registrul FLAGS (EFLAGS or RFLAGS), în octetul cel mai puţin semnificativ conţinutul registrului AH, adică:

FLAGS ← $[AH]_{0-7}$

Instrucţiunea nu are operanzi.

Ex.

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

Instrucţiunea PUSH

Instrucţiunea PUSH decrementează registrul ESP şi copie operandul sursă în stivă. Un operand pe 16 biţi decrementează registrul ESP cu 2, iar un operand pe 32 biţi – cu 4. Sunt 3 formate ale instrucţiunii:

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

Instrucţiunea POP

Instrucţiunea PUSH copie conţinutul stivei în operandul sursă pe 16 sau 32 biţi şi incrementează registrul ESP cu valorile 2 sau 4 respectiv. Sunt 2 formate ale instrucţiunii:

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

Instrucţiunile PUSHFD şi POPFD

Instrucţiunea PUSHFD copie conţinutul registrului EFLAGS pe 32 biţi în stivă, iar POPFD extrage din stivă valoarea pe 32 biţi şi încarcă registrul de fanioane EFLAGS.

```
pushfd
popfd
```

Instrucţiunile PUSHAD, PUSHA, POPAD şi POPA

Instrucţiunea PUSHAD copie conţinutul registrelor de uz general în stivă în ordinea următoare:

EAX, ECX, EDX, EBX, ESP (valorile înaintea execuţiei PUSHAD), EBP, ESI şi EDI. Iar inctrucţiunea POPAD extrage din stivă şi încarcă regiştrii de uz general în ordinea inversă.

Respectiv instrucţiunea PUSHA copie conţinutul registrelor de uz general pe 16 biţi în stivă în ordinea următoare: AX, CX, DX, BX, SP, BP, SI, DI. Iar inctrucţiunea POPA extrage din stivă şi încarcă regiştrii de uz general pe 16 biţi, în ordinea inversă.

2.7.2 Aritmetica binara

Aceste instrucțiuni modifica conţinutul registrului FLAGS.

a) Instrucţiunea ADD (Add)

Forma generală:

ADD dest, sursa ; [dest] ← [dest] + [sursa]

unde:

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

Operanzii au aceeaşi structură ca la instrucţiunea MOV. Cei doi operanzi nu pot fi simultan locaţii de memorie.

Operaţia se poate efectua pe 8,16, 32 sau pe 64 biţi. Cei doi operanzi trebuie sa aibă aceeaşi dimensiune (acelaşi tip). În caz de ambiguitate se va folosi operatorul PTR.

Indicatorii afectaţi sunt: AF, CF, PF, SF, ZF şi 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) Instrucţiunea ADC (Add with Carry)

Forma generală:

ADC dest, sursa ; [dest) ← [dest] + [sursa] + [CF]

Unde *dest* şi *sursa* au aceeasi semnificaţie ca la instrucţiunea ADD, iar CF este Carry Flag.
Instrucţiunea adună conţinutul *dest* cu conţinutul *sursei* și cu bitul de transport CF. Indicatorii afectaţi sunt aceeaşi de la instrucţiunea ADD.

Operaţia ADC se folosește la adunări de operanzi pe mai multe cuvinte, operaţie în care poate apărea transport de care trebuie să se țină seama.

Exemplu 1. Să se adune doua 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 biţi (FFh + FFh), producând un rezultat pe 16-biţi, suma va fi in DL:AL, care este 01FEh:

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

c) Instrucţiunea SUB (Substrat)

Forma generală:

SUB dest, sursa ; [dest] ← [dest] – [sursa]

unde *dest* şi *sursa* au aceeaşi semnificaţie ca la instrucţiunea ADD. Indicatorii afectaţi sunt cei specificaţi la ADD. Structura operanzilor ca la instrucţiunea MOV.

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

d) Instrucţiunea SBB (Substrat with Borrow)

Forma generală:

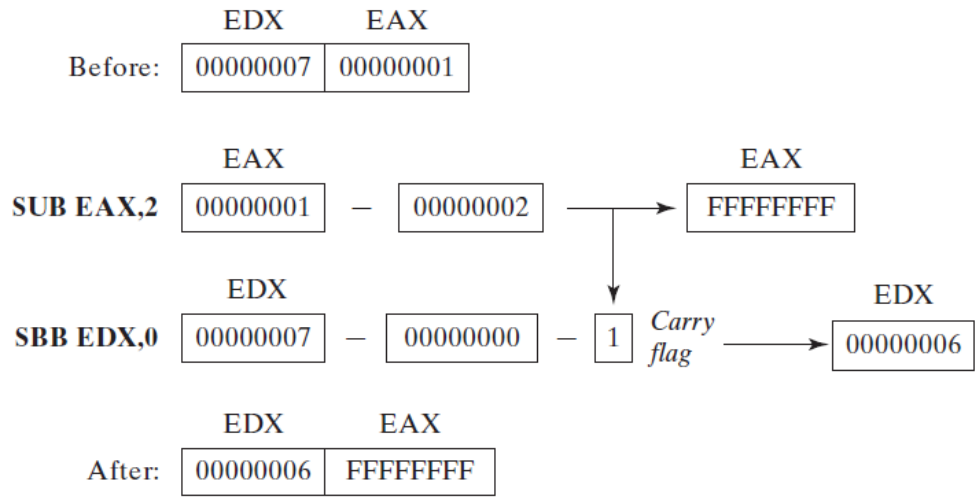
SBB dest, sursa ; [dest] ← [dest] – [sursa] – [CF]

unde semnificaţia *dest*, *sursa* şi CF sunt cele prezentate la ADC. Instrucţiunea 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 biţi un untreg de 32 biţi. Perechea de regiştri EDX:EAX se încarcă cu valoarea 0000000700000001h şi 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 inderioară
sub eax,2 ; scăderea  2
sbb edx,0 ; scăderea părţii superioare
```



e) Instrucţiunea INC (Increment)

Forma generală:

INC dest ; [dest] ← [dest] + 1

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

Exemple:

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

f) Instrucţiunea DEC (decrement)

Forma generală:

DEC dest ; [dest] ← [dest] – 1

unde *dest* are aceeaşi semnificaţie ca *dest* de la Instrucţiunea INC. Aceeaşi indicatori ca la INC sunt afectaţi.

g) Instrucţiunea NEG (Negate)

Forma generală:

NEG dest : [dest] ← 0 – [dest] schimbare de semn

unde *dest* este un operand pe 8, 16, 32 sau 64 biţi ce poate fi un registru general sau o locaţie de memorie. Instrucţiunea afectează indicatorii AF, CF, PF, SF, OF şi ZF.

Exemplu:

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

h) Instrucţiunea CMP (Compare)

Forma generală:

CMP dest, sursa ; [dest] – [sursa]

Instrucţiunea realizează o operaţie de scădere intre cei doi operanzi, fără a modifica operandul *dest* sau *sursa* cu poziţionarea indicatorilor de condiţie. Indicatorii afectaţi sunt: AF, CF, PF, SF, ZF şi OF.

Aceasta instrucţiune se foloseşte împreuna cu instrucţiunea de salt condiţionat.

Când comparăm doi operanzi fără semn indicatoarele Zero şi Carry indică următoarea relaţie între operanzi:

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

Exemple:

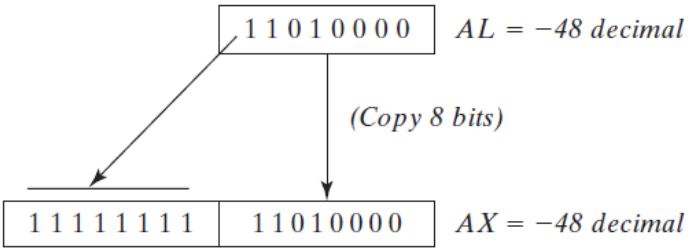
```
cmp     ax, alfa[bx][si]
cmp      [si], 0
```

i) Instrucţiunea CBW (Convert Byte to Word)

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

daca bitul de semn $AL_7 = 0$ atunci [ah] ← 00h

altfel [ah] ← 0ffh.



Instrucţiunea nu are operanzi şi nu afectează indicatorii de condiţie.
Exemplu. Se cere să se adune un număr întreg cu semn reprezentat pe octet cu un număr întreg cu semn pe cuvânt.

```
a       sbyte   -75
b       sword   -188
c       sword   ?
. . . . . .
mov     al, a
cbw     ;               converteşte octetul la cuvânt
add     ax, b
mov     c, ax
. . . . . .
```

j) Instrucţiunea CWD (Convert Word to Double Word)

Are ca efect extinderea bitului de semn din AX [AX_{15}] la întreg registrul DX, obţinându-se astfel AX pe 32 de biţi, adică:

daca semnul [AX_{15}] = 0 atunci [dx] ← 0000h,
altfel [dx] ← 0ffffh.

Instrucţiunea nu are operanzi şi nu afectează indicatorii de condiţie.
Exemplu. Se cere diferenţa dintre un operand reprezentat pe 2 cuvinte (do) şi unul reprezentat pe cuvânt (so)

```
do      dword  12345678h
so      word   0abcdh
rez     dword  ?
. . . . . .
mov     ax, so
cwd             ; operandul so reprezentat în DX : AX
mov     bx, ax  ; salvează ax în bx
mov     ax, word  ptr do
sub     ax, bx
mov     word  ptr  rez, ax
mov     ax, word  ptr  do + 2
sbb     ax, dx  ; ia în considerare eventualul transport
mov     word  ptr  rez + 2
```

Instrucţiunea CDQ (convert doubleword to quadword)

Are ca efect extinderea bitului de semn din EAX [EAX_{15}] la întreg registrul EDX, obţinându-se astfel EAX pe 64 de biţi, adică:

daca semnul [EAX_{15}] = 0 atunci [EDX] ← 00000000h,
altfel [edx] ← 0ffffffffh.

Instrucţiunea nu are operanzi şi nu afectează indicatorii de condiţie.

k) Instrucţiunea MUL (Multiply)

Forma generală:

```
MUL reg/mem8
MUL reg/mem16
```

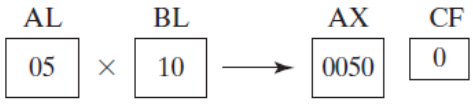
```
MUL reg/mem32
```

unde *reg* poate fi un registru sau o locaţie de memorie *mem* de 8, 16, 32 biţi. Rezultatul se obţine pe un număr dublu de biţi (16, 32, 64). Operaţia realizată este produsul intre acumulator şi sursa cu depunerea rezultatului în acumulatorul extins. Cei doi operanzi se consideră numere fără semn.
Dacă *sursa* este pe octet avem:

[AX] ← [AL] * [*reg/mem8*]

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

Diagrama ilustrează interacţiunea dintre registre:



Dacă sursa este pe cuvânt avem:

[DX:AX] ← [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 interacţiunea dintre registre:

Dacă sursa este pe 32 biţi avem:

[EDX:EAX] ← [EAX] * [*reg/mem32*]

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

Diagrama ilustrează interacţiunea dintre registre:

iar dacă sursa este pe 64 biţi avem:

[RDX:RAX] ← [RAX] * [*reg/mem64*]

```
mov rax,0FFFF0000FFFF0000h
mov rbx,2
mul rbx ; RDX:RAX = 0000000000000001FFFE0001FFFE0000
```

Afectează indicatorii CF şi OF, ceilalţi sunt nedefiniţi.

l) Instrucţiunea IMUL (Integer Multiply)

Instrucţiunea IMUL semnifică înmulţirea cu semn. Instrucţiunea poate avea 1, 2, sau 3operanzi.

Afectează indicatorii CF și OF, restul sunt nedefiniţi.

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 lăţimea registrului de destinaţie. Dacă cifrele semnificative sunt pierdute, se setează indicatorii CF şi 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) Intructiunea DIV (Divide)

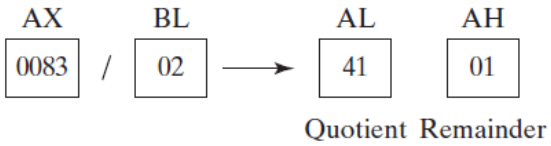
Forma generală:

DIV sursa

unde *sursa* este un registru sau o locaţie de memorie, reprezentata pe octet, cuvânt, 32 biţi sau 64 biţi. Instrucţiunea realizează împărţirea fără semn intre deîmpărţit şi împărţitor. Daca împărţitorul (sursa) este reprezentat pe octet atunci deîmpărţitul este AX şi rezultatul este: câtul în **al** iar restul în **ah**, adică:

DIV sursa ; [al] ←[ax]/ [sursa]
; [ah] ←restul împărţirii [ax]/ [sursa]

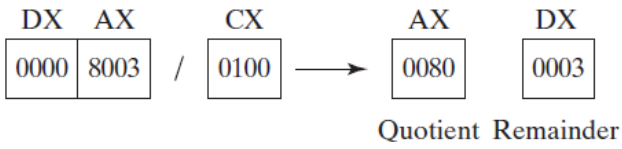
```
mov ax,0083h     ; deîmpărţitul
mov bl,2         ; împărţitorul
div bl      ; AL = 41h-catul, AH = 01h-restul
```



Daca împărţitorul (sursa) este reprezentat pe cuvânt atunci deîmpărţitul este considerat în DX şi AX, câtul se obţine în AX iar restul în DX, adică

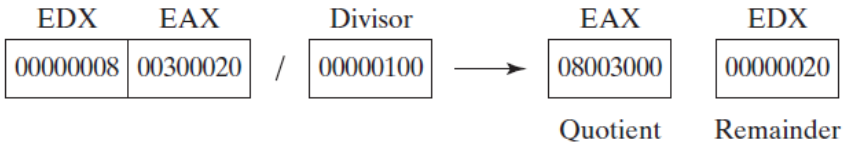
DIV sursa ; [ax] ←câtul împărţirii [dx:ax]/[sursa]
; [dx] ←restul împărţirii [dx:ax]/[sursa]

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



Daca împărţitorul (sursa) este reprezentat pe 32 biţi atunci deîmpărţitul este considerat în EDX şi EAX (64 biţi), câtul se obţine 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
```



Daca împărţitorul (sursa) este reprezentat pe 64 biţi atunci deîmpărţitul este considerat în RDX şi RAX (64 biţi), câtul se obţine în RAX iar restul în RDX

```
.data
dividend_hi QWORD 0000000000000108h
dividend_lo QWORD 0000000033300020h
divisor QWORD 0000000000010000h
.code
mov rdx,dividend_hi
mov rax,dividend_lo
div divisor     ; RAX = 0108000000003330
                ; RDX = 0000000000000020
```

Toţi indicatorii nu sunt definiţi. Operaţia de împărţire poate conduce la depăşiri, dacă câtul depăşeşte valoarea maximă reprezentabilă pe 8, respectiv pe 16 biţi sau daca împărţitorul este 0.

n) Instrucţiunea IDIV (Integer Divide)

Forma generală:

IDIV sursa

Semnificaţia instrucţiunii şi a operandului *sursa* este aceeasi ca la Instrucţiunea DIV, cu o singură diferenţă importantă – deîmpărţitul cu semn trebuie să fie extins înainte ca împărţirea să fie executată.
Extinderea semnului se execută cu instrucţiunile ***CBW, CWD, CDQ.***
Indicatorii sunt nedefiniţi. Operaţia poate conduce la depăsiri.
Ex. Împărţim -48 la +5

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

2.7.3 Aritmetica BCD

2.3.1 Instrucţiunea AAA (ASCII Adjust for Addition)

Instrucţiunea nu are operanzi şi execută corecţia acumulatorului AX, după operaţii de adunare cu numere în format BCD despachetat. Semnificaţia este:

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

Indicatorii afectaţi : AF, CF, restul nedefiniţi.
Exemplu:

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

2.3.2 Instrucţiunea AAS (ASCII Adjust for Subtraction)

Instrucţiunea nu are operanzi şi execută corecţia acumulatorului AX, după operaţii de scădere cu numere in format BCD despachetat. Semnificaţia este:

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

Indicatorii afectaţi : AF, CF, restul nedefiniţi.
Exemplu:

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

2.3.3 Instrucţiunea DAS (Decimal Adjust for Substraction)

Instrucţiunea nu are operanzi şi execută corecţia zecimala a acumulatorului AL, după operaţii de scădere cu numere în format BCD împachetat. Semnificaţia este:

daca $[AL_{0:3}] > 9$ 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 destinaţie
        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 doreşte o
                                        ; adresare la nivel de octet
        mov    esi,2
        mov    ebx,3
        mov    alfa[ebx][esi],33h  ; Adresare bazata indexata
                                ; relativa a operandului destinaţie
        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 Instrucţiunea *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 – cuvant, iar b, c, g, h, i –byte. Ca să putem executa împărţirea cu **f** convertim împărţitorul la dublucuvânt. Ne vor interesa doar câturile împărţirilor, 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 câtul în ax  si restul în 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 în al si restul în ah
mov rez, al

exit
main ENDP
END main
```

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

Fereastra consolei (sau fereastra – linie de comandă, cmd.exe) este o fereastră textuală, creată de MS-Windows, pentru afişarea liniei de comandă. Vom descrie unele funcţii utilizate pentru a introduce date de la tastatură şi afişarea datelor de ieşire. Aceste proceduri se apelează cu instrucţiunea *call,* de exemplu *call Clrscr.*

Procedurile apelate utilizează diferite echivalări.

2.7.5 Echivalări şi operatori

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

a) Echivalările numerice sunt folosite pentru a atribui o constanta numerica 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 şir 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 redefinita în timpul asamblării.

Sintaxa unei echivalări numerice neredefinibile este:

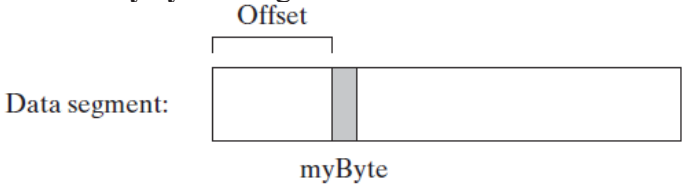
nume EQU **expresie**

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 admişi in cazul folosiri expresiilor constante sunt operatori aritmetici: +, -, *, /(împărțire întreagă) şi 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 octeţi.

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

SIZEOF. Operatorul SIZEOF întoarce numărul total de octeți alocați pentru un tablou sau variabilă definita 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 sfârşitul liniei, din rând nou
Delay	Întrerupe execuția programului pentru un interval specificat de milisecunde
DumpMem	Procedura afişează un tablou de date (array) în hexazecimal
DumpRegs	Afişează conţinutul regiştrilor EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, şi 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
ReadString	Citește de la tastatură un șir de caractere, finalizarea introducerii - tasta Enter
SetTextColor	Setează culorile textului și de fundal a consolei.
WaitMsg	Afișează un mesaj și așteaptă un clic pe o tastă
WriteBin	Afișează un întreg fără semn pe 32 de biți în format binar ASCII
WriteBinB	Afișează un întreg în format binar pe un octet, cuvânt, sau 32 de biți
WriteChar	Afișează un singur caracter
WriteDec	Afișează un întreg fără semn pe 32 de biți în format zecimal
WriteHex	Afișează un întreg pe 32 de biți în format hexazecimal
WriteHexB	Afișează un întreg de un byte, word, sau doubleword în format hexazecimal
WriteInt	Afișează un întreg cu semn pe 32 de biți în format zecimal
WriteString	Afișează un șir, finalizat cu un octet nul
WriteWindowsMsg	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 - încărcaţi deplasamentul tabloului, în ECX - numărul de locaţii de memorie, iar în EBX – tipul locaţiei 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 locaţii de memorie
mov ebx,TYPE array ; tipul locaţiei de memorie - doubleword
call DumpMem
```

La ieșire se va afișa:

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

DumpRegs. Afișează conţinutul regiştrilor EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, şi registrul EIP în hexazecimal. Se afișează, de asemenea, valorile fanioanelor (indicatoarelor) Carry, Sign, Zero, Overflow, Auxiliary Carry, şi 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 apelaţi Gotoxy, încărcaţi 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 aşteaptă un singur caracter introdus de la tastatură şi îl încarcă în registrul AL. Caracterul este introdus fără ecou (fără afişare).

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

Dacă executaţi clic pe o tastă funcţională (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-biţi fără semn de la tastatură și încarcă valoarea în EAX. Spatiile 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 si 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). Spaţiile sunt ignorate.

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

ReadInt. Procedura ReadInt citește un întreg de 32-biţi, 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-biţi (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 "ABCDEFG" în variabila *buffer* va fi:

```
41 42 43 44 45 46 47 00          ABCDEFG
```

Conţinutul variabilei `byteCount` va fi egală cu 7.

SetTextColor. Procedura *SetTextColor* setează culorile textului și de fundal a consolei. La invocarea procedurii culorile textului şi 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 secvenţă 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. Biţii sunt afişaţi în grupe de câte 4 biţi.

```
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. Încărcaţi în EAX întregul şi în EBX indicaţi valoarea de afişat în octeţi (1, 2, sau 4). Biţii sunt afişaţi în grupe de câte 4 biţi.

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

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

WriteChar. Procedura WriteChar afișează un singur caracter. Încărcaţi caracterul de afişat (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. Încărcaţi întregul în EAX.

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

WriteHex. Procedura *WriteHex* afișează un întreg pe 32 de biți în format hexazecimal. Introduceţi zerouri, dacă este necesar. Încărcaţi 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. Încărcaţi în EAX întregul şi în EBX indicaţi valoarea de afişat în octeţi (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 introdiceţi semnul. Încărcaţi întregul în EAX.

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

WriteString. Procedura *WriteString* afișează un şir, finalizat cu un octet nul. Încărcaţi offset-ul şirului în registrul EDX.

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

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

```
call WriteWindowsMsg
```

Un exemplu de mesaj:

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

2.7.7 Instrucţiuni de salt și ciclare

a)Instrucţiunea de salt necondiţionat JMP

Forma generala :

JMP operand

unde, operand este adresa de salt necondiţionat. Există următoarele tipuri de instrucţiuni JMP:

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

b)Instrucţiuni de salt condiţionat

Aceste instrucţiuni implementează salturile condiţionate de indicatorii de condiţie.

Forma generala:

Jcond operand

unde:

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

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

Se observă că există 2 categorii de instrucţiuni pentru 'mai mic' si 'mai mare', cele care conţin cuvintele 'above' sau 'bellow' şi cele care conţin cuvintele 'less' sau 'greater'. Primele se folosesc în situaţia comparării a două valori fără semn, iar ultimele în situaţia comparării a două valori cu semn.

Fie secvenţele de program:

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

şi

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

în care se compară pe cuvânt 0FFFEh şi 2.

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

Fiecare mnemonică din tabel se referă la iniţialele cuvintelor următoare, ce indică condiţia 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 (depăşire de capacitate), Parity (PEven - paritate pară, POdd - paritate impară), Sign (semn).

Instrucţiune (mnemonica)	Condiţie de salt	Interpretare
JE, JZ	ZF = 1	Zero, Equal
JL, JNGE	SF ≠ OF	Less, Not Greater or Equal
JLE,JNG	SF ≠ 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 si ZF = 0	Not Less or Equal, Greater
JNB, JAE, JNC	CF = 0	Not Below, Above or Equal, Not Carry
JNBE, JA	CF = 0 si 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) Instrucţiunea JCXZ (JUMP if CX is Zero)

Instrucţiunea realizează salt la eticheta specificată dacă conţinutul registrului CX este zero. Forma generala:

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 toţi biţii (2, 3 şi 7) sunt setaţi 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 şi RCX în modurile pe16 şi 64 biţi respectiv) şi dacă acesta este diferit de zero se sare la eticheta specificată, în caz contrar se continuă cu instrucţiunea următoare. D8 este un deplasament pe 8 biţi şi reprezintă diferenţa între offset-ul instrucţiunii următoare instrucţiunii LOOP şi offset-ul etichetei. Se utilizează şi notaţiile:

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

Ex: Suma celor n octeţi 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) Instrucţiunea LOOPZ/LOOPE (LOOP While Zero/Equal)

Forma generala**:**

LOOPZ eticheta

sau

LOOPE eticheta

Semnificaţia:

cx ← cx - 1
daca cx > 0 si ZF = 1 atunci
[IP] ← [IP] + D8

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

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

Forma generala:

LOOPNZ eticheta

sau

LOOPNE eticheta

Semnificaţia:

cx ←cx - 1
daca cx > 0 şi ZF = 0 atunci
[IP] ← [IP] + D8

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

2.7.8 Instrucţiuni de deplasare (SHL, SAL, SHR, SAR) si de rotaţie (ROL, RCL, ROR, RCR).

Acest grup de instrucţiuni realizează operaţii de deplasare şi de rotaţie la nivel de bit. Instrucţiunile au doi operanzi: primul este operandul propriu-zis, iar al doilea este numărul de biţi cu care se deplasează sau se roteşte primul operand. Ambele operaţii 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 şi 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 generala 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.

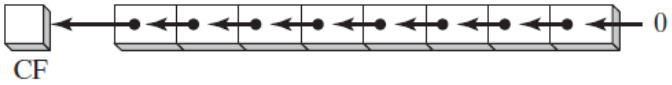
La operațiile de deplasare , se modifică toate flag-urile conform rezultatului, în afară de AF, care este nedefinit. La operațiile de rotație, se modifică numai CF şi OF.

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

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

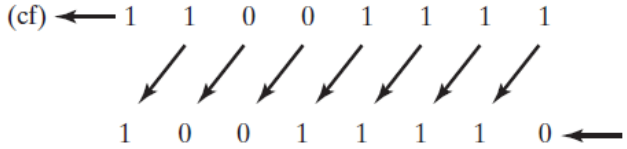
Are forma generală:

SHL/SAL operand, contor



Deşi există două mnemonice (SHL şi 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 instrucţiunii (similar şi SHR, SAL, SAR, ROR, ROL, RCR, RCL) este următoarea:

```
SHL reg,imm8
SHL mem,imm8
SHL reg,CL
SHL mem,CL
```

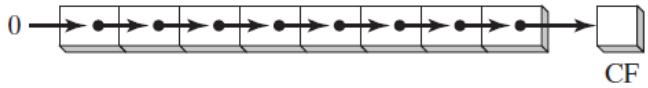
Exemple:

```
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 in 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).

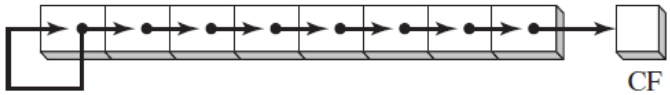
Exemple:

```
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 in 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).

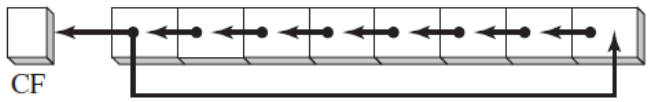
Exemple:

```
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 ax,-128 ; EAX = ????FF80h
shl eax,16 ; EAX = FF800000h
sar eax,16 ; EAX = FFFFFF80h
```

d) Instrucțiunea ROL (Rotate Left)

Are forma generală:

ROL operand, contor



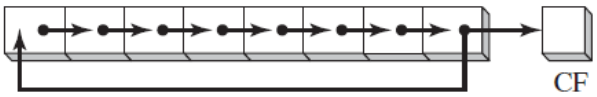
Bitul cel mai semnificativ din *operand* trece atât in CF, cât şi în bitul cel mai puțin semnificativ din operand, după ce toți biții acestuia s-au deplasat la stânga cu o poziție. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

```
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 generala:

ROR operand, contor



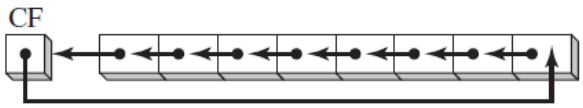
Bitul cel mai puțin semnificativ din *operand* trece atât în CF, cât şi în bitul cel mai semnificativ din operand, după ce toți biții acestuia s-au deplasat la dreapta cu o poziție. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

```
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



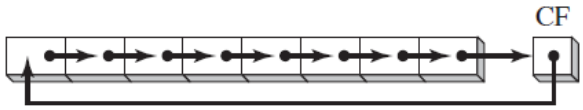
Bitul cel mai semnificativ din *operand* trece în CF, se deplasează toți biții din operand cu o poziție la stânga, iar CF inițial trece în bitul cel mai puțin semnificativ din operand. Operația se repetă de atâtea ori cât este valoarea lui *contor* (0-255 sau conținutul registrului CL).

```
clc ; 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:

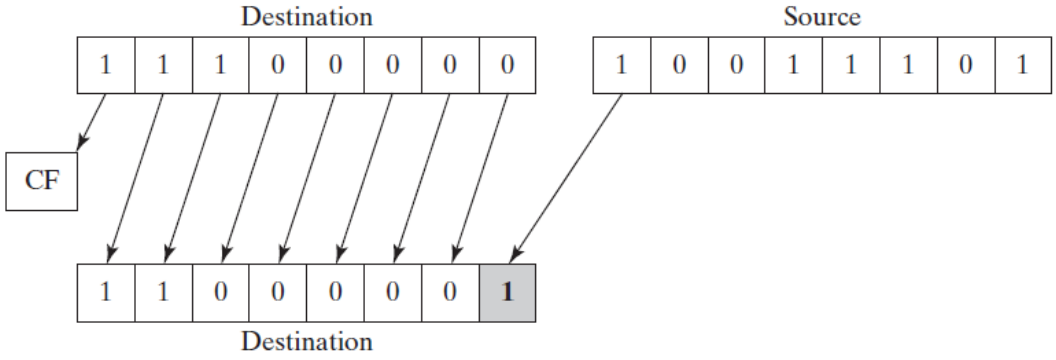
SHLD *dest, sursa, contor*

SHRD *dest, sursa, contor*

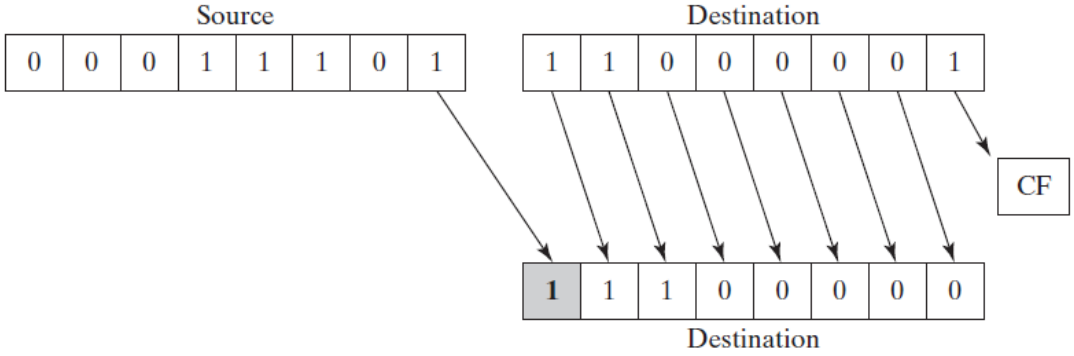
Structura instrucţiunii:

```
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:

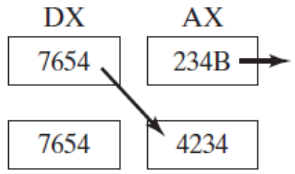


Iar deplasarea dreapta după următoarea schemă:



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 biţi. Operațiile se fac la nivel de bit.

a) Instrucțiunea NOT (Not)

Forma generala :

NOT dest

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

operandului, adică se face complementul faţă de unu.

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

b) Instrucţiunea AND (And)

Forma generala :

AND dest, sursa

în care *dest* poate fi un registru sau o locaţie de memorie, iar *sursa* un registru, o locaţie de memorie sau o constantă. Instrucţiunea depune în *dest* şi-logic la nivel de bit între *dest* şi *sursa*.
Indicatori afectaţi: SF, ZF, PF, CF=0, OF=0, AF nedefinit. Structura instrucţiunii:

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

Valoarea imediată *imm* nu poate depăşi mărimea de 32 biţi.

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

c) Instrucţiunea OR (Or)

Forma generala :

OR dest, sursa

în care *dest* poate fi un registru sau o locaţie de memorie, iar *sursa* un registru, o locaţie de memorie sau o constanta. Instrucţiunea depune în *dest* sau-logic la nivel de bit între *dest* şi *sursa*.
Indicatori afectaţi: SF, ZF, PF, CF=0, OF=0, AF nedefinit.

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

d) Instrucţiunea XOR (Exclusive Or)

Forma generala :

XOR dest,sursa

în care *dest* poate fi un registru sau o locaţie de memorie, iar *sursa* un registru, o locaţie de memorie sau o constantă. Instrucţiunea depune în *dest* xor-logic la nivel de bit între *dest* şi *sursa*.
Indicatori afectaţi: SF, ZF, PF, CF=0, OF=0, AF nedefinit. Tabelul de adevăr este următorul:

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

e) Instrucţiunea TEST (Test)

Forma generala :

TEST dest,sursa

în care *dest* poate fi un registru sau o locaţie de memorie, iar *sursa* un registru, o locaţie de memorie sau o constanta. Instrucţiunea realizează and-logic la nivel de bit între *dest* şi *sursa* şi nu modifică destinaţia, cu poziţionarea indicatorilor.
Indicatori afectaţi: SF, ZF, PF, CF=0, OF=0, AF nedefinit.
Această instrucţiune este deseori utilizată pentru a testa starea unui bit (biţi).
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)Instrucţiunile BSF, BSR (Bit scan forward, reverse)

Forma generala :

BSF dest,sursa

```
BSF reg16,r/m16
BSF reg32,r/m32
```

- Instrucţiunea scanează biţii 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 întâlneşte bitul de 1, flag-ul ZF este setat în 0, iar în operandul destinaţie este încărcat indexul (poziţia) 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:
```

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

Forma generala :

BT dest,sursa

```
BT r/m16,imm8
BT r/m16,r16
BT r/m32,imm8
BT r/m32,r32
```

Instrucţiunea BT copie bitul ***n,*** specificat în sursă, în flag-ul Carry. Operandul destinaţie conţine valoarea cu bitul căutat, iar operandul sursă conţine poziţia bitului căutat.

Instrucţiunea BTC copie bitul ***n,*** în flag-ul Carry şi compementează bitul ***n*** în operandul destinaţie.

Instrucţiunea BTR copie bitul ***n,*** în flag-ul Carry şi setează bitul ***n*** în 0 în operandul destinaţie.

Instrucţiunea BTS copie bitul ***n,*** în flag-ul Carry şi setează bitul ***n*** în 1 în operandul destinaţie.

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

Exemple programe

{Căutare secvenţiala} Să se scrie un program pentru căutarea primului blanc din şirul începând de la adresa **sir**. La ieşirea din program e**ax** va conţine valoarea 0 daca şirul nu conţine blancuri, altfel va conţine valoarea poziţiei din sir a primului blanc găsit. Se presupune că şirul **sir** are **l** 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 anunţă
;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 şi posibilitatea efectuării unor operații de transfer, sau operații aritmetice şi logice cu şiruri de date (cu informații aflate în zone continue de memorie). Operațiile pe şiruri pot fi efectuate individual, pentru fiecare cuvânt din şir, sau automat - cu repetare, numărul de repetări al instrucțiunii fiind dictat de conținutul unui registru contor.

Operațiile tipic efectuate sunt:

- transferul unui şir din zonă sursa în zonă destinaţie
- comparare între două şiruri
- căutarea unei valori într-un şir
- încărcarea acumulatorului cu elementele unui şir.
- citirea unui şir de la un port de intrare
- scrierea unui şir la un port de ieşire

Exemple :

Instrucţiunile MOVSB (Move (copy) bytes)
MOVSW (Move (copy) words)
MOVSD (Move (copy) doublewords)

Transfer pe 8 (16,32) biţi, 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 operaţii cu şiruri sunt utilizate prefixe de repetare:

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

Exemplu. Fie dat să copiem 20 de cuvinte duble din şirul sursă **source** în şirul destinaţie **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  soursei
mov edi,OFFSET target ; incarcam EDI cu adresa destinaţiei
     rep movsd           ;copiem cuvinte duble
```

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

Comparare pe 8 (16,32) biţi, 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 şi EDI în mod automat, până când ECX =0 sau o pereche de cuvinte duble nu va fi egală.

Instrucţiunile SCASB (SCAS- Scans a string)
SCASW
SCASD

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

Instrucţiunile STOSB (STOS- Store string data)
STOSW
STOSD

Instrucţiunile 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. Şirul `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
```

Instrucţiunile LODSB (LODS- Load Accumulator from String)
LODSW
LODSD

Instrucţiunile încarcă valoarea din byte, word sau doubleword din memorie idicat de ESI, în AL/AX/EAX respectiv. Instrucţiunile sunt utile la căutarea unui singur element într-un şir.
Exemplu: Multiplicarea fiecărui element a unui şir 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 şi macroinstruţiuni

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* şi *FAR* lipsesc, în cazul utilizării definiţiilor complete se consideră implicit *NEAR*.

Tehnicile de transfer a parametrilor combină diversele modalităţi de alegere a tipurilor de locaţii fizice pentru păstrarea parametrilor transmişi: registre, locaţii 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.

***Instructiunea 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 şi FAR. Tipul apelului trebuie să coincidă cu tipul procedurii şi cu tipul instructiunii RETURN din interiorul procedurii.

Instructiunea RET (RETURN)

Forma generală:

RET [n]

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

Dacă instrucțiunea RET este de tip NEAR semnificația sa este:

[IP] ← SS: [[SP] + 1: [SP]]

[SP] ← [SP] + 2

[[SP] ← [SP] + n]

adică se reface (IP) prin copierea conținutului vârfului stivei şi incrementarea cu 2 a lui (SP). Dacă în instrucțiunea RET apare şi 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 numar in DX:AX
    mov  ax,WORD PTR a
    mov  dx,WORD PTR [a+2]
        ; incarca al doilea number in DI:SI
    mov  si,WORD PTR b
    mov  di,WORD PTR [b+2]
; incarca adresa rezultatului in BX
    mov  bx,OFFSET s
; apeleaza 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 secvența de cod, între directivele MACRO si 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, blancuri 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, blancuri 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 macroinstructiuni, care la fiecare apel al macroinstructiunii 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 macrodefinitie, trebuie să urmeze imediat directivei MACRO.

Exemplu: Ridicarea unui numar 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 funcţionale şi clasificarea

Noţiunea de la care pornim este aceea de calculator; acesta este un sistem programabil de prelucrare a informaţiei care are două componente esenţiale, inseparabile şi definitorii: *hardware şi software.*

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

Blocurile funcţionale sunt:

1. *Unitatea centrală de prelucrare* (UCP) are două funcţii esenţiale:
   - prelucrarea datelor;
   - controlul activităţii întregului calculator.

O unitate centrală de prelucrarea informaţiei, având funcţiile enunţate mai sus, care coordonează un sistem structurat funcţional ca în Figura 3.1 şi care, fizic, se prezintă sub forma unui singur cip (circuit integrat) se numeşte *microprocesor*. Această accepţiune standard a noţiunii va fi folosită în continuare pentru detalierea conceptelor care stau la baza funcţionării întregului calculator.

2. *MEMORIA* este, din punctul de vedere al sistemului pe care îl definim, o secvenţă de locaţii pentru stocarea informaţiei.

Fiecare locaţie este definită prin două entităţi informaţionale:

- *Conţinutul,* reprezentat de o înşiruie de cifre binare 0 sau 1 ("biţi"); se va observa că nu am folosit noţiunea de "număr binar", pentru că informaţia stocată într-o locaţie de memorie poate avea diverse semnificaţii. Numărul de cifre binare conţinute într-o locaţie depinde de modul în care microprocesorul organizează informaţia în memorie; mărimea unei locaţii va fi denumită *formatul memoriei,* exprimat în număr de biţi (8 biţi). Formatul memoriei nu are nici-o legătură cu *organizarea fizica* a cipurilor de memorie**.**

Figura 3.1

- *Adresă,* reprezentând numărul de ordine al locaţiei, care permite identificarea sa în cadrul secvenţei de locaţii (există o corespondenţă biunivocă între fiecare locaţie de memorie şi adresa sa).
- În privinţa memoriei unui calculator vom folosi câteva noţiuni:
- *"Harta memoriei",* definită ca fiind totalitatea locaţiilor de memorie pe care le poate adresa un microprocesor.
- *"Pagini"* şi/sau *"segmente"* sunt subdiviziuni *logice* ale hărţii 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/ ieşire* (I/O*)* sunt constituite din circuitele prin care se realizează legătura între calculator şi lumea exterioară. O unitate elementară de conversaţie cu exteriorul poartă numele de *"port de intrare/ieşire"*. Între porturi şi locaţiile din memorie există nişte similitudini:

- Porturile sunt în esenţă tot locaţii de memorare a informaţiei, adresabile; desigur, informaţia care se foloseşte uzual aici este alcătuită din operanzi/rezultate (date).
- Există o *"hartă a porturilor"* care, aşa cum vom arăta în capitolele următoare, poate sau nu să facă parte din harta memoriei.

Singura deosebire esenţială faţă de locaţiile de memorie este legătura fizică pe care porturile o asigură cu exteriorul*;* pentru microprocesor, de multe ori, această legătură fizică este transparentă şi nesemnificativă.

În sfârşit, componenţa hardware a calculatorului comportă un set de legături specifice; acestea se realizează printr-o aşa numită "magistrală": un set de conexiuni fizice între blocuri prin care informaţia care circulă are o semnificaţie prestabilită. Sistemele la care ne referim au o magistrală unică, ce le caracterizează; din punct de vedere funcţional, există trei componente ale acestei magistrale, individualizate şi în Figura 3.1:

1. *Magistrala de date,* bidirecţională, permite circulaţia datelor (operanzi/rezultate), a instrucţiunilor şi chiar a adreselor.

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

3. *Magistrala de control* permite circulaţia, bidirecţională, a semnalelor de comandă şi 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 noţiuni şi definirea lor ne permit câteva concluzii care să facă o primă delimitare asupra conceptului de microprocesor aşa cum este el înţeles în volumul de faţă:

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

- Toate celelalte componente ale sistemului nu au putere de decizie. Memoria, de pildă, nu controlează şi nici nu e necesar să controleze semnificaţia informaţiei pe care o deţine şi modul în care este organizată logic.
- Legătura dintre blocuri este asigurată de o magistrală unică cu trei componente funcţionale; pe magistrala de date circulă toate tipurile de informaţii.
- Funcţionarea sistemului se face pe baza unor programe alcătuite din secvenţe de instrucţiuni. Acestea sunt citite din memorie de către microprocesor, recunoscute şi 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) Lăţimea magistralei de date – lăţimea (mărimea) maximală a codului informatic care poate fi prelucrat, păstrat şi transferat în calculator, ca o unitate întreagă.

b) Capacitatea de stocare a memoratoarelor – cantitatea informaţiei codate, concomitent păstrată în memoratoarele calculatorului. Pentru măsurarea capacităţii de stocare se folosesc următoarele prefixe:

- 1KiloByte = 2^{10}Byte ≈ 10^3 Byte
- 1MegaByte = 2^{20} Byte ≈ 10^6 Byte
- 1GigaByte = 2^{30} Byte ≈ 10^9 Byte
- 1TerraByte = 2^{40} Byte ≈ 10^{12}Byte
- 1PetaByte = 2^{50} Byte ≈ 10^{15}Byte

Clasificarea calculatoarelor conform caracteristicilor menţionate este neactuală.

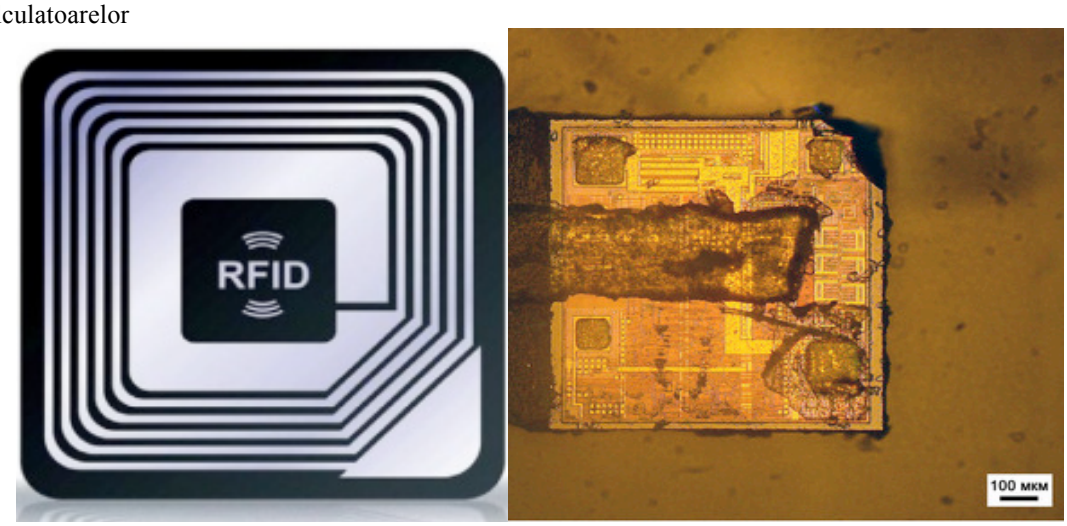
Prezentăm câteva clasificări moderne.

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, maşini, diferite dispozitive;
– console de jocuri – jocuri la domiciliu;
– calculatoare personale PC – variantele desktop și laptop-uri;
– servere - servere de rețea;
– grupe (clastere) de staţii 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 radiofrecvenţă). Această tehnologie presupune stocarea informaţiilor nu prin codurile de bare, ci prin intermediul unor cipuri electronice integrate, de ex. în etichete, ecusoane, ambalaje de marfă, corpurile animalelor etc. Aceste informaţii, 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 incorporează 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 incorporat este (aproape) transparentă pentru utilizator.

Printre multele 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, incorporate intr-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 spatiu 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,

Wi-Fi şi Ethernet.

Calculatoare personale

Calculatoare personale (PC) se divizează în 2 grupe: variantele desktop şi 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 deservesc 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 (clastere) de staţii de lucru (COW- Cluster Of Workstations)

Clasterele constau din zeci, sute, mii de PC-uri sau statii de lucru conectate in retea prin placi de retea de larg consum. Sistemele COW sunt gestionate de soft specializat, ce permite să direcționeze resursele lor pentru a rezolva diferite probleme inginerești 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 clastere de stații de lucru.

Calculatoarele Mainframe

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

Supercalculatorul posedă resurse hardware şi software deosebite. Se utilizează în industria de apărare, în cercetarea ştiinţifică, în câteva universităţi, în industria aeronautică şi spaţială. Departamentul Energiei SUA deţine un supercomputer din lume *The Roadrunner*. Acesta are o putere de calcul de 1 petaflop (10^{15} operaţii pe secundă, în virgulă mobilă). Ocupă o suprafaţă de 1100 m2 şi 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 operaţii 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 existenţa unei metode de comparare a diferitelor arhitecturi, fără a fi necesară compararea specificaţiilor 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 urmăreşte fluxul de date şi de instrucţiuni prin acestea. Prin flux de instrucţiuni se înţelege secvenţa de instrucţiuni executată de o maşină sau unitate de execuţie; iar prin flux de date se înţelege secvenţa de date apelate de fluxul de instrucţiuni.

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

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

SISD (Single Instruction Single Data):

Din această categorie fac parte calculatoarele convenţionale care execută un singur flux de instrucţiuni asupra unui singur flux de date. Aceste siteme de calcul se mai numesc şi calculatoare von Neumann.

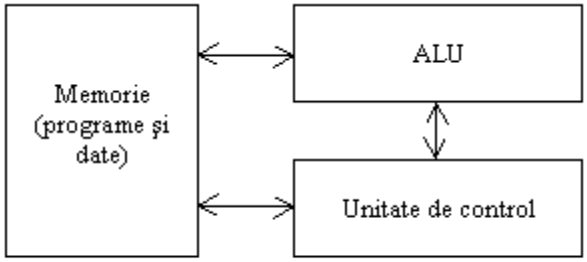


Figura 3.2 Arhitectura von Neumann

Instrucţiunile sunt executate secvenţial, 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 unităţi funcţionale (ex: coprocesor matematic, procesor grafic, procesor de intrare/ieşire, etc.), însă acestea sunt văzute ca o singură unitate de execuţie.

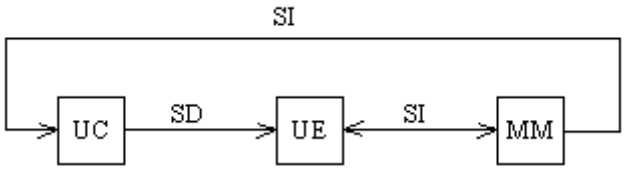


Figura 3.3 Arhitectura SISD

UC – unitate de comandă;
UE – unitate de execuţie, element de procesare, procesor;
MM – modul de memorie;
SI – flux (şir) de instrucţiuni;
SD – flux (şir) 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 unităţi de execuţie identice aflate sub comanda unei singure unităţi de control. Unitatea de control transmite acelaşi flux de instrucţiuni, simultan, tuturor unităţilor de execuţie. Toate unităţile de execuţie execută simultan aceeaşi instrucţiune asupra datelor din memoria proprie (există sisteme ce au şi o memorie partajată pentru comunicaţii). Unitatea de control trebuie să permită tuturor elementelor de procesare să-şi termine instrucţiunea curentă înainte de iniţierea unei noi instrucţiuni, astfel că execuţia instrucţiunilor trebuie sincronizată între toate unităţile de execuţie. Ca şi ordin de mărime numărul procesoarelor implicate într-o structură SIMD este de câteva mii.

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

asupra datelor. Exemplu: reţele 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.

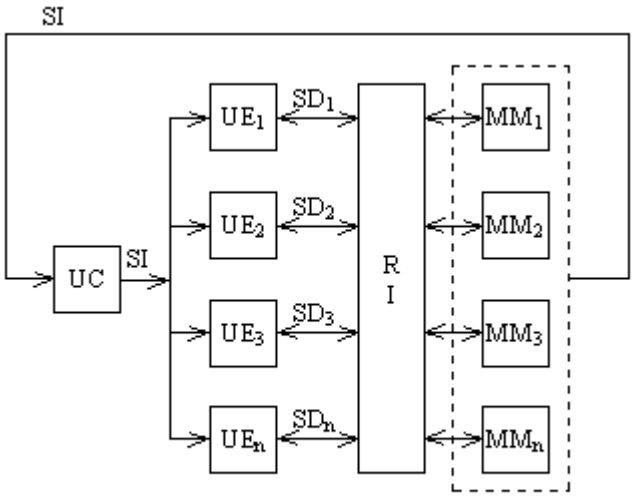


Figura 3.4 Arhitectura SIMD

Topologia reţelei 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 instrucţiuni asupra unui singur flux de date. Acest lucru este realizabil în două moduri:

- acelaşi element din fluxul de date este prelucrat de toate procesoarele, fiecare executând propriile operaţii asupra respectivei date;
- un element din fluxul de date este prelucrat de primul procesor, rezultatul obţinut este pasat mai departe celui de-al doilea procesor ş.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 şi poate opera în modurile SIMD, MISD şi MIMD.

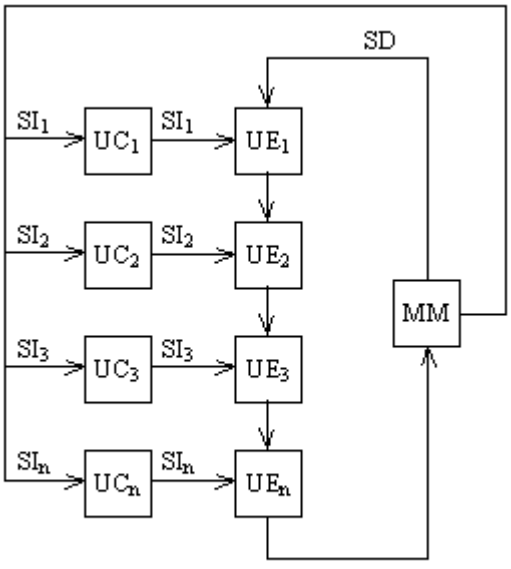


Figura 3.5 Arhitectura MISD

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 instrucţiuni. Sistemele MIMD pot avea şi memorie partajată. Operaţiile executate de fiecare procesor sunt independente intre ele, deci modul lor de operare este asincron.

Acest tip de arhitecturi sunt aplicabile în cazul aplicaţiilor 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.

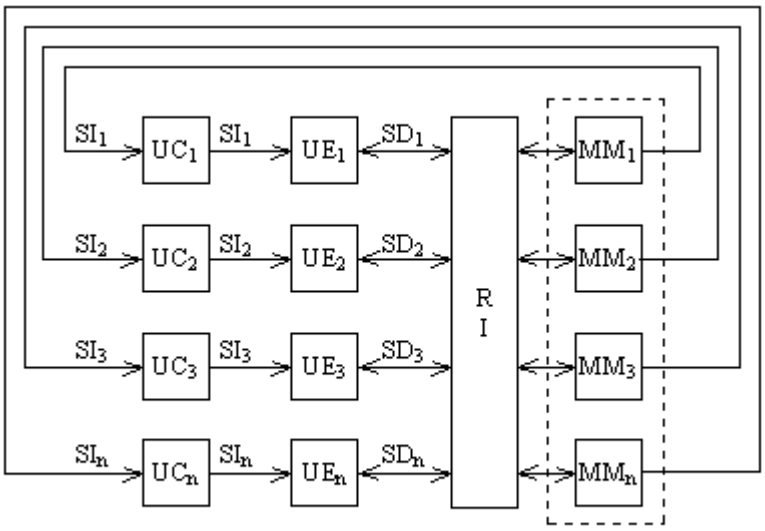


Figura 3.6 Arhitectura MIMD

O variantă între SIMD şi MIMD sunt arhitecturile **SPMD** (Single Program Multiple Data), în care unităţile de procesare execută acelaşi 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 şi 8008, pe patru şi respectiv 8 biţi. Primul microprocesor considerat "standard", care impune chiar o definire a termenului şi a unor concepte legate de această modalitate de prelucrare a informaţiei este însă 8080 produs tot de firma Intel. Tot firma Intel este cea care lansează primul microprocesor care lucrează pe 16 biţi - 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 biţi. Strategia firmei este limpede: mulţi fabricanţi nu sunt pregătiţi să schimbe toate celelalte componente ale sistemelor de prelucrare pe 16biţi, aşa că vor prefera încă microprocesoarele compatibile cu magistrala de 8 biţi.
În tabelul de mai jos sunt prezentate generaţiile şi carateristicile de bază ale microprocesoarelor Intel.

Tabelul 1 - Generaţiile şi carateristicile de bază ale microprocesoarelor Intel

Tip/generaţie	Anul	Lăţimea magistralei Date/Adrese, Biţi	Cache Interior (L1), kB	Frecvenţa de tact a magistralei de memorie, (Mhz)	Frecvenţa de tact (interioară,) (Mhz)
8088/First 8086/First	1979 1978	8/20 16/20	None None	4.77-8 4.77-8	4.77-8 4.77-8
80286/Second	1982	16/24	None	6-20	6-20
80386DX/Third 80386SX/Third	1985 1988	32/32 16/32	None None	16-33 16-33	16-33 16-33
80486DX/Fourth 80486SX/Fourth 80486DX2/Fourth 80486DX4/Fourth	1989 1989 1992 1994	32/32 32/32 32/32 32/32	8 8 8 8+8	25-50 25-50 25-40 25-40	25-50 25-50 50-80 75-120
Pentium/Fifth Pentium MMX/Fifth	1993 1997	64/32 64/32	8+8 16+16	60-66 66	60-200 166-233
Pentium Pro/Sixth Pentium II/Sixth	1995 1997	64/36 64/36	8+8 16+16	66 66	150-200 233-300
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, anunţă primele calculatoare personale, IBM PC-XT, care folosesc 8088/8086. Aceste procesoare au introdus conceptul de segmentare a memoriei: memoria este împărţită 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 funcţiona numai în *modul rea*l, care este un mod uniproces, în care se execută un singur proces (program sau *tas*k) 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, deşi tot pe 16 biţi, introduce o serie de noi concepte fundamentale care tind să schimbe chiar noţiunea de microprocesor. Procesorul 80286 poate funcţiona în *modul real* al procesoarelor precedente, dar dispune şi de un mod de adresare virtual sau *mod protej*at. Acest mod utilizează conţinutul 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 funcţiona în regim multi-proces *(multitaskin*g), în care pot fi executate mai multe procese în mod concurent. În acest mod se realizează o comutare prin hardware între procesele care se execută concurent. Firma IBM lansează şi ea o nouă generaţie de calculatoare personale: PC-AT ("Advanced Technology") care foloseşte Intel 80286.

Procesorul 80386 a introdus în cadrul arhitecturii *Intel* registre de 32 de biţi, utilizate atât pentru păstrarea datelor, cât şi pentru adresare. Pentru compatibilitate cu procesoarele anterioare, aceste registre s-au obţinut prin extinderea registrelor de 16 biţi, fiind posibilă utilizarea în continuare a vechilor registre, acestea constituind jumătatea de ordin inferior a registrelor de 32 de biţi. A fost introdus un nou mod de funcţionare, numit *mod virtual 808*6. Instrucţiunile originale au fost extinse cu noi forme care utilizează operanzi şi adrese de 32 de biţi, şi au fost introduse instrucţiuni complet noi, ca de exemplu instrucţiuni pentru operaţii 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 şi o memorie încorporată *cache* cu informaţii despre până la 32 de pagini cel mai recent accesate.

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

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

În 1993, se lansează primul microprocesor al generaţiei a cincea, numit din acest motiv Pentium. Procesorul *Pentium* a adăugat o nouă linie de execuţie de tip *pipeline* a instrucţiunilor, pentru a se obţine performanţe superscalare. Cele două linii de execuţie a instrucţiunilor, numite U şi V, permit execuţia a două instrucţiuni pe durata unei perioade de tact. Capacitatea memoriei *cache* L1 a fost de asemenea dublat , fiind alocaţi 8 KB pentru instrucţiuni şi 8 KB pentru date. Memoria *cache* pentru date utilizează protocolul MESI, care permite gestiunea memoriei *cache* atât prin metoda mai eficientă *"write-back*", cât şi prin metoda *"write-through"* utilizată de procesorul 80486. Procesorul *Pentium* utilizează predicţia salturilor pentru a creşte performanţele construcţiilor care utilizează bucle de program. Registrele generale sunt tot de 32 de biţi, dar s-au adăugat magistrale interne de date de 128 şi 256 de biţi pentru a creşte viteza transferurilor interne, iar magistrala externă de date a fost extinsă la 64 de biţi. 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 şi 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 îmbunătăţită, care permite execuţia a trei instrucţiuni într-o stare (perioadă de tact, ceas). Procesorul *Pentium Pr*o, ca şi următoarele procesoare din familia P6, se caracterizează prin execuţia dinamică a instrucţiunilor, care constă din analiza fluxului de date, execuţia instrucţiunilor într-o altă ordine decât cea secvenţială, o predicţie îmbunătăţită a salturilor şi execuţia speculativă. Pe lângă cele două memorii *cache* L1 de câte 8 KB, prezente şi la procesorul *Pentiu*m, procesorul *Pentium Pro* dispune şi de o memorie *cache* L2 de 256 KB, aflat în acelaşi circuit cu unitatea centrală , conectat cu aceasta printr-o magistrală dedicată de 64 de biţi. Procesorul *Pentium Pro* are o magistrală de adrese extinsă la 36 de biţi, astfel încât spaţiul adreselor fizice este de până la 64 GB.

Procesorul *Pentium* II se bazează pe arhitectura *Pentium Pr*o, la care s-au adăugat extensiile MMX *(Multimedia Extension*s). Memoria *cache* L2 a fost mutată în afara capsulei procesorului. Atât memoria *cache* L1 pentru date, cât şi memoria *cache* L1 pentru instrucţiuni 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"* şi *"Deep Sleep*", pentru reducerea puterii consumate în perioadele de inactivitate.

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

Primul din familia P7, numit Pentium 4, a primit şi o nouă arhitectură cu o viteză de procesare mai performantă. În versiunea microprocesorului cu frecvenţa 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 măreşte viteza de procesare. Pentru creşterea vitezei de prelucrare a datelor audio şi video a fost introdus un set suplimentar din SSE instrucţiuni.

În noembrie 2004 firma Intel a renunţat la producerea în serie a microprocesorului Pentium 4 cu frecvenţa de 4 Ghz din cauza dificultăţilor apărute la răcirea lui.

Actualmente, firma Intel, încapsulează două şi mai multe nuclee de microprocesor cu frecvenţe reduse în circuitul unui microprocesor, ce exclud dificultăţile cu răcirea lor.

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

Există însă şi alte direcţii de dezvoltare a dispozitivelor de prelucrare a informaţiei; un exemplu sunt aşa numitele "procesoare cu set redus de instrucţiuni" (RISC) având ca reprezentanţi 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) şi altele.

3.2 Microprocesoarele CISC/RISC

Multe microprocesoare au seturi de instrucţiuni ce includ mai mult de 100 – 200 instrucțiuni. Ele folosesc o varietate de tipuri de date şi 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 operanzi din memorie;
- Format de *lungime variabilă* pentru instrucțiuni;
- Unitate de control microprogramată (micro-codată), avantajoasa 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 performanţele codului obiect;

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

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

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

Aşa cum s-a arătat mai sus, arhitecturile RISC restricţionează numărul de instrucţiuni care accesează direct memoria principală. Cele mai multe instrucţiuni ale RISC presupun doar operaţii între registrele interne UCP. Pentru că instrucţiunile complexe nu există în setul de instrucţiuni, dacă este nevoie de ele, acestea se implementează prin rutine cu ajutorul instrucţiunilor existente. În final, într-un program executabil vor fi mai multe instrucţiuni decât la CISC, dar execuţia 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. Instrucţiunile microprocesoarelor x86 de tip CISC, înainte de executarea lor, sunt transformate într-un set simplu de instrucţiuni 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 noţiunea de *arhitectura suprascalară.* Pentru a explica această noţiune, urmărim utilizarea microinstrucţiunilor 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:

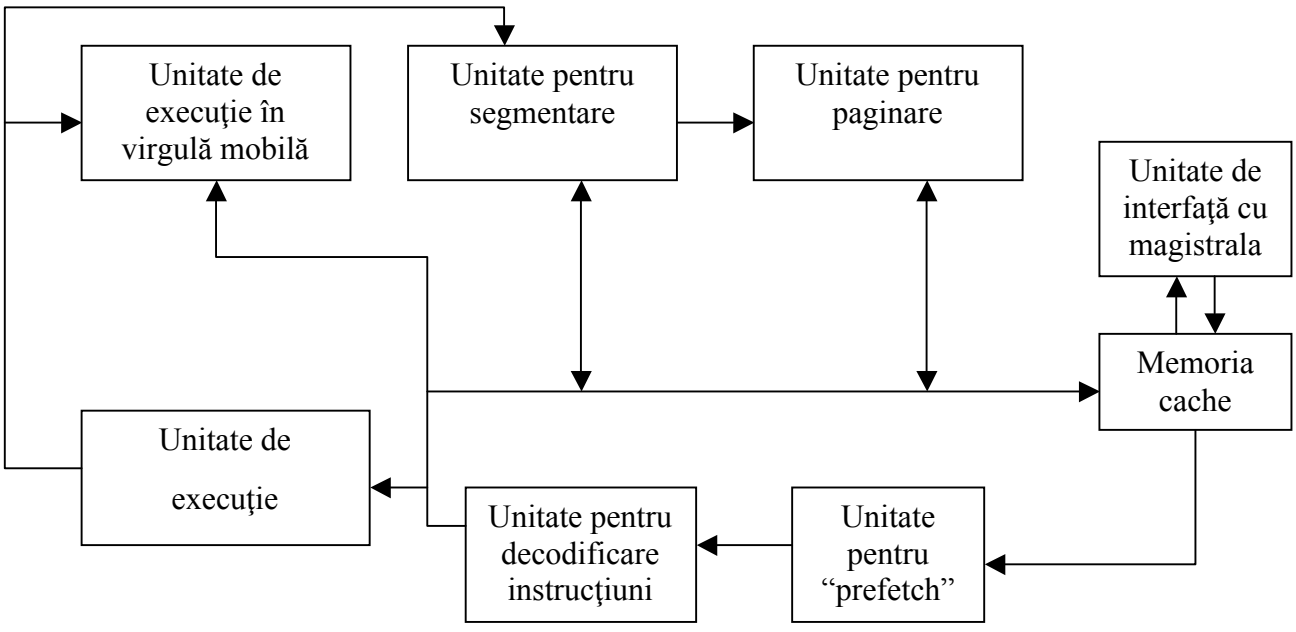


Figura 3.7

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

Toate aceste segmente sunt executate în paralel. Din unitatea „prefetch” instrucţiunea se transferă în unitatea pentru decodificarea instrucţiuni, şi unitatea „prefetch” este liberă şi poate citi următoarea instrucţiune. Deci, în interiorul microprocesorului se află 5 instrucţiuni în diferite segmente de execuţie. 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 operanzii. Segmentul 3 localizează și extrage operanzii, fie din regiștri, fie din memorie. Segmentul 4 execută instrucțiunea, de obicei rulând operanzii prin calea de date, iar segmentul 5 scrie rezultatul în regiștri.

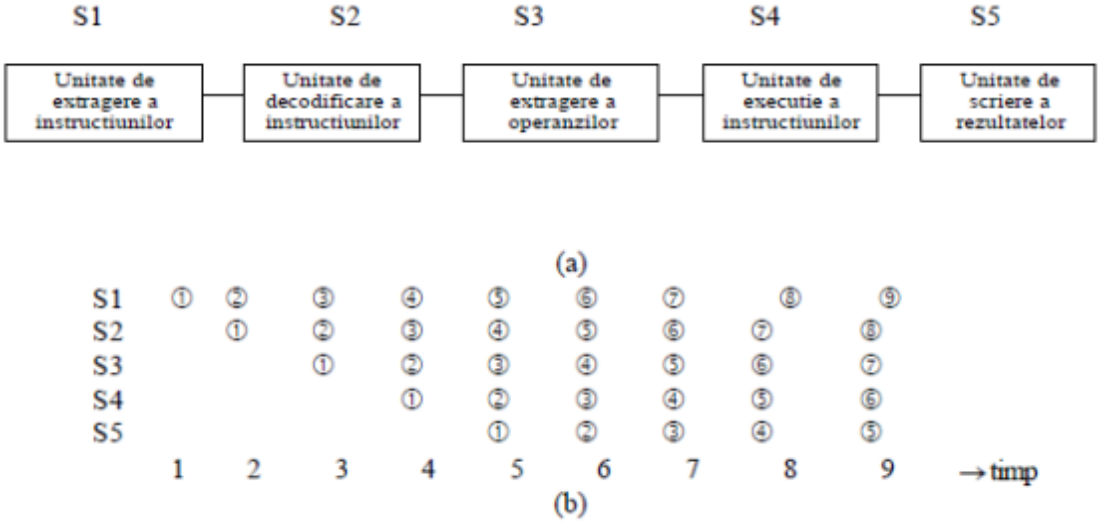


Figura 3.8 - O banda de asamblare de 5 segmente (a)
Starea fiecărui segment în funcție de timp (b).

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 operanzii pentru instrucțiunea 1, S2 decodifica 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 şi poate executa 2 instrucţiuni 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 dual㠍. 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 anterioara e respectată, fie conflictele sunt detectate și eliminate pe parcursul execuției, cu ajutorul unui hardware suplimentar.

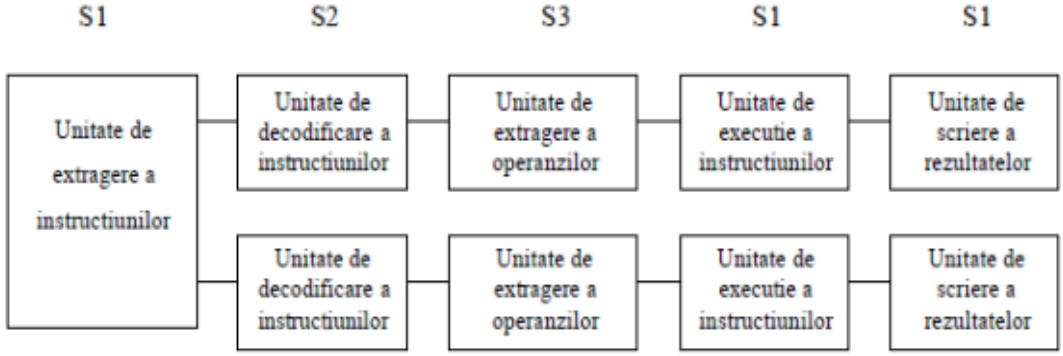


Figura 3.9 Banda de asamblare dual㠍 cu 5 segmente

Pentium are două benzi de asamblare asemănătoare cu cele din fig. 3.9, deși împărțirea între segmentele 2 si 3 (numite decode-1 si decode-2) este puțin diferită față de cea din exemplul nostru. Banda de asamblare principală, numita ”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. Idea 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.

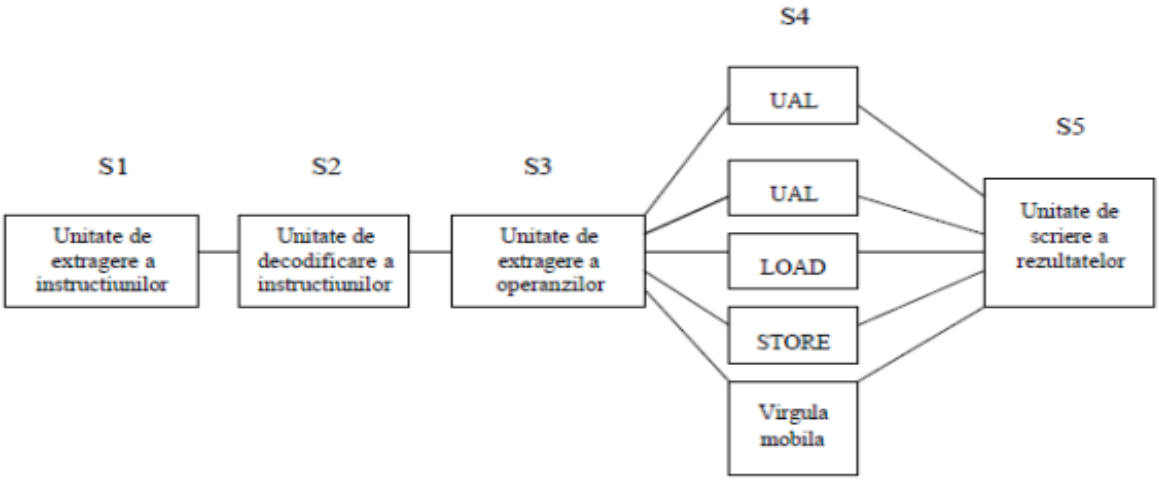


Figura 3.10 - Un procesor superscalar cu 5 unități funcționale

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.