# 5. SETURI DE INSTRUCȚIUNI

Funcționarea UCP este determinată de instrucțiunile mașină pe care le execută. Colecția diferitelor instrucțiuni pe care le poate executa UCP reprezintă *setul de instrucțiuni* al UCP.

## 5.1. Elementele unei instrucțiuni mașină

Fiecare instrucțiune trebuie să conțină informațiile necesare pentru execuția acesteia de către UCP. Acestea sunt:

- *Codul operației*: Reprezintă un cod binar care specifică operația de executat.
- *Adresa operandului sursă*: Operația poate necesita unul sau mai mulți operanzi sursă, deci operanzi asupra cărora se efectuează operația.
- *Adresa rezultatului*: Operația poate produce un rezultat.
- *Adresa instrucțiunii următoare*: Indică adresa de la care se va extrage instrucțiunea următoare, după terminarea execuției instrucțiunii curente.

În cele mai multe cazuri, următoarea instrucțiune care se va executa urmează imediat după instrucțiunea curentă. În aceste cazuri, nu există o referință explicită la următoarea instrucțiune.

Modul în care sunt specificate adresele datelor și instrucțiunilor va fi prezentat în secțiunea legată de modurile de adresare.

Operanzii sursă și rezultatul se pot afla în:

- *Memoria principală sau secundară*.
- *Registru* UCP: Fiecărui registru i se atribuie un număr unic (o adresă), iar instrucțiunea trebuie să conțină acest număr.
- *Dispozitiv de I/E*: Instrucțiunea trebuie să specifice modulul de I/E și dispozitivul utilizat în cadrul operației.

## 5.2. Limbaje de asamblare

Pentru scrierea instrucțiunilor mașină de către programatori se utilizează o *reprezentare simbolică*. Codurile de operație se reprezintă prin nume simbolice, numite *mnemonice*. De exemplu:

| | |
|---|---|
| `ADD` | Adunare |
| `SUB` | Scădere |
| `MUL` | Înmulțire |
| `DIV` | Împărțire |
| `LOAD` | Încărcarea datei din memorie |
| `STORE` | Memorarea datei |

Considerăm următoarea instrucțiune într-un limbaj de nivel înalt:

$N = I + J + K$

Pentru scrierea acestei instrucțiuni utilizând o reprezentare simbolică, presupunem că variabilele *I*, *J* şi *K* se inițializează cu 2, 3, respectiv 4. Fragmentul de program va începe de la adresa `100h`, iar pentru variabile se rezervă un spațiu de memorie începând de la adresa `200h`. Reprezentarea simbolică a fragmentului este prezentată în continuare.

```
100    LOAD        200
101    ADD         201
102    ADD         202
103    STORE       203

200    DATA        2
201    DATA        3
202    DATA        4
203    DATA        0
```

Fiecare linie constă din trei câmpuri. Primul conține adresa unei locații de memorie. Pentru o instrucțiune, al doilea câmp conține mnemonicul instrucțiunii. Pentru o instrucțiune cu referire la memorie, al treilea câmp conține adresa. Pentru reprezentarea datelor în memorie, se utilizează o *pseudoinstrucțiune* cu numele `DATA`. Aceasta indică faptul că al treilea câmp al liniei conține o valoare hexazecimală care va fi memorată în locația specificată de primul câmp.

Această formă de scriere are dezavantajul că trebuie cunoscute adresele absolute ale instrucțiunilor şi datelor. Aceasta înseamnă ca programul şi datele nu pot fi încărcate decât într-o singură zonă de memorie, care trebuie cunoscută dinainte. În plus, dacă se modifică ulterior programul, de exemplu prin adăugarea unei noi linii, se vor modifica toate adresele din liniile următoare.

Pentru a se elimina aceste dezavantaje, adresele instrucțiunilor şi a datelor sunt de asemenea reprezentate simbolic. Acelaşi fragment de program se poate scrie astfel:

```
CALCUL LOAD        I
       ADD         J
       ADD         K
       STORE       N
```

```
I       DATA        2
J       DATA        3
K       DATA        4
N       DATA        0
```

Primul câmp conține o adresă simbolică, numită *etichetă*. Unele linii nu au o adresă, ceea ce presupune că adresa liniei respective este cea imediat următoare a liniei precedente.

Această scriere a instrucțiunilor şi a datelor reprezintă un *limbaj de asamblare*. Programele scrise într-un asemenea limbaj sunt translatate în limbaj maşină cu ajutorul unui program numit *asamblor*. Asamblorul trebuie să translateze mnemonicele în coduri binare de instrucțiuni, şi să atribuie adrese numerice adreselor simbolice.

## 5.3. Numărul de adrese ale instrucțiunilor

Fiecare instrucțiune este împărțită în câmpuri, care conțin elementele componente ale acesteia. Această organizare a instrucțiunii se numeşte *formatul instrucțiunii*. Cele mai multe seturi de instrucțiuni utilizează mai multe formate de instrucțiuni. Un exemplu simplu este prezentat în Figura 5.1.

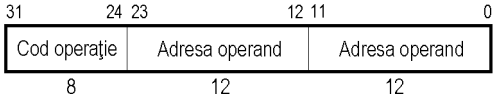


**Figura 5.1.** Un format simplu de instrucțiuni.

Acest format utilizează *două adrese*, fiind utilizat de exemplu pentru instrucțiunile aritmetice şi logice care necesită doi operanzi. Pentru memorarea rezultatului, poate fi necesară o a *treia adresă*. După execuția unei instrucțiuni, trebuie încărcată următoarea instrucțiune, iar adresa acesteia poate fi de asemenea specificată în cadrul instrucțiunii. Deci, o instrucțiune poate conține patru adrese.

În practică, instrucțiunile cu patru adrese se utilizează rareori. Cele mai multe seturi utilizează una, două sau trei adrese, adresa următoarei instrucțiuni fiind implicită (obținută din contorul de program).

În Figura 5.2 se prezintă fragmente de program pentru calculul expresiei:

$$Y = (A - B) / (C + D * E)$$

utilizând instrucțiuni tipice cu una, două, respectiv cu trei adrese.

```
LOAD   D            ; AC ← D
MUL    E            ; AC ← AC * E
ADD    C            ; AC ← AC + C
STORE  Y            ; Y ← AC
LOAD   A            ; AC ← A
SUB    B            ; AC ← AC - B
DIV    Y            ; AC ← AC / Y
STORE  Y            ; Y ← AC
```

a) Instrucțiuni cu o adresă

```
MOV    Y,A          ; Y ← A
SUB    Y,B          ; Y ← Y - B
MOV    T,D          ; T ← D
MUL    T,E          ; T ← T * E
ADD    T,C          ; T ← T + C
DIV    Y,T          ; Y ← Y / T
```

b) Instrucțiuni cu două adrese

```
SUB    Y,A,B        ; Y ← A - B
MUL    T,D,E        ; T ← D * E
ADD    T,T,C        ; T ← T + C
DIV    Y,Y,T        ; Y ← Y / T
```

c) Instrucțiuni cu trei adrese

**Figura 5.2.** Fragmente de program pentru calculul expresiei
$Y = (A - B) / (C + D * E)$

Dacă se utilizează instrucțiuni *cu o adresă*, a doua adresă este implicită, al doilea operand fiind păstrat în registrul acumulator AC. Rezultatul se obține de asemenea în acumulator.

Pentru instrucțiunile *cu două adrese*, una din adrese are rol dublu, indicând atât unul din operanzi, cât şi rezultatul. Astfel, instrucțiunea

```
SUB    Y,B
```

efectuează operația $Y - B$ şi memorează rezultatul în $Y$. Numărul de instrucțiuni necesare pentru scrierea fragmentului de program este mai redus decât în cazul formatului cu o adresă. Pentru a evita modificarea valorii unui operand, se utilizează o instrucțiune `MOV` pentru a transfera una din valori în locația rezultatului sau într-o locație temporară $T$.

În cazul formatului cu *trei adrese*, numărul de instrucțiuni necesare se reduce, dar creşte lungimea fiecărei instrucțiuni. De aceea, acest format este utilizat mai rar.

Anumite instrucțiuni nu necesită nici o adresă. Acest format, numit cu *zero adrese*, poate fi utilizat mai ales în cazul instrucțiunilor cu memoria stivă. Ca operanzi

se vor utiliza două elemente din vârful stivei, deci de la adresa indicată de SP, respectiv SP - 1.

Tabelul 5.1 prezintă modul de interpretare al instrucțiunilor cu 0, 1, 2 şi 3 adrese. În fiecare caz, se presupune că adresa următoarei instrucțiuni este implicită, şi se execută o operație de adunare între doi operanzi.

**Tabelul 5.1.** Interpretarea instrucțiunilor în funcție de numărul de adrese.

| Număr de adrese | Reprezentare simbolică | Interpretare |
|---|---|---|
| 0 | ADD | (SP) ← (SP) + (SP - 1) |
| 1 | ADD A | AC ← AC + A |
| 2 | ADD A, B | A ← A + B |
| 3 | ADD A,B,C | A ← B + C |

Un număr mai mic de adrese pe instrucțiune are ca rezultat instrucțiuni mai simple, ceea ce necesită un UCP mai puțin complex, şi instrucțiuni cu o lungime mai mică. Pe de altă parte, programele vor conține un număr mai mare de instrucțiuni, spațiul de memorie necesar fiind mai mare.

În cazul instrucțiunilor cu o singură adresă, programatorul dispune de obicei de un singur registru general, acumulatorul. La instrucțiunile cu mai multe adrese, de obicei există registre generale multiple. Aceasta permite ca anumite operații să fie executate numai în registre, ceea ce creşte viteza de execuție. Cele mai multe calculatoare utilizează o combinație de instrucțiuni cu două şi trei adrese.

## 5.4. Tipuri de instrucțiuni

Tipul instrucțiunilor disponibile variază de la un calculator la altul. Totuşi, se pot deosebi următoarele categorii generale întâlnite la majoritatea calculatoarelor:

- Transferuri de date;
- Instrucțiuni aritmetice;
- Instrucțiuni logice;
- Instrucțiuni de salt şi de apel;
- Instrucțiuni de control al sistemului;
- Instrucțiuni de I/E.

### 5.4.1. Transferuri de date

O instrucțiune de transfer trebuie să specifice în primul rând adresa sursei şi cea a destinației transferului. Sursa şi destinația poate fi memoria, un registru sau vârful stivei. De asemenea, trebuie să se indice lungimea datelor de transferat şi modul de adresare al sursei şi al destinației.

Pot exista coduri de operație diferite pentru transferul între două registre sau între un registru şi memorie, caz în care vor exista mnemonice diferite pentru cele două categorii. O altă posibilitate este ca tipul operandului (registru sau memorie) să fie indicat prin modul de adresare, caz în care va exista un singur mnemonic. De multe ori, nu se admit doi operanzi de memorie, deci nu se poate realiza transferul direct între două locații de memorie.

Pentru transferul datelor de diferite lungimi (8, 16, 32 sau 64 biți) pot exista coduri diferite sau acelaşi cod, în ultimul caz lungimea datelor fiind precizată într-un câmp separat al instrucțiunii.

Din punctul de vedere al operațiilor executate de UCP, instrucțiunile de transfer sunt dintre cele mai simple. Dacă atât sursa cât şi destinația sunt registre, UCP execută o operație internă de transfer între cele două registre. Dacă unul sau ambii operanzi sunt în memorie, UCP trebuie să calculeze adresa de memorie, pe baza modului de adresare.

### 5.4.2. Instrucțiuni aritmetice

Cele mai multe calculatoare dispun de instrucțiuni cu numere întregi (*în virgulă fixă*) pentru operațiile aritmetice elementare: adunare, scădere, înmulțire şi împărțire. Pot exista şi instrucțiuni pentru numere *în virgulă mobilă* sau *zecimale* (în cod BCD).

Alte operații posibile sunt diferite operații cu un singur operand (unare), de exemplu:

- Calculul valorii absolute;
- Complementare (schimbarea semnului operandului);
- Incrementare cu 1;
- Decrementare cu 1.

### 5.4.3. Instrucțiuni logice

Aceste instrucțiuni permit operații asupra biților individuali ai unui octet sau cuvânt. Aceste operații se bazează pe logica booleană. Cele mai uzuale operații logice sunt:

- ŞI logic (AND);
- SAU logic (OR);
- SAU EXCLUSIV (XOR);
- Complement (NOT).

În Tabelul 5.2 se prezintă exemple pentru aceste operații, efectuate între conținutul a două registre R1 şi R2.

**Tabelul 5.2.** Exemple de operații logice cu operanzi aflați în registrele R1 şi R2.

| | |
|---|---|
| R1 | 1010 0110 |
| R2 | 0000 1111 |
| R1 AND R2 | 0000 0110 |
| R1 OR R2 | 1010 1111 |
| R1 XOR R2 | 1010 1001 |
| NOT R1 | 0101 1001 |

Operația ŞI logic se poate utiliza ca o mască pentru selecția anumitor biți ai cuvântului şi ştergerea celorlalți biți. Similar, operația SAU logic se poate utiliza pentru setarea la 1 a anumitor biți, fără modificarea celorlalți biți. Operația SAU EXCLUSIV se poate utiliza pentru complementarea anumitor biți. Dacă toți biții unuia din operanzi sunt 1, celălalt operand va fi complementat față de 1.

Pe lângă instrucțiunile care permit operații la nivel de bit, există de obicei diferite instrucțiuni pentru *deplasare* şi *rotire*. Operațiile de deplasare şi rotire la dreapta sunt ilustrate în Figura 5.3. Există operații similare pentru deplasarea şi rotirea la stânga.

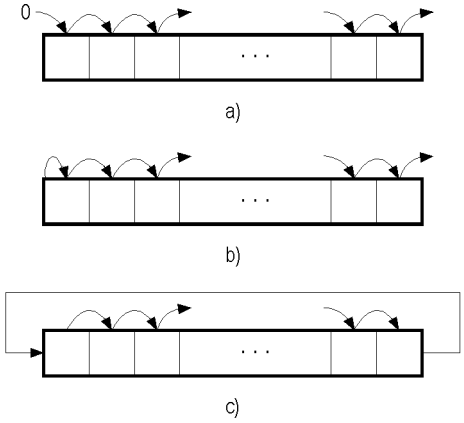


**Figura 5.3.** Ilustrarea operațiilor de deplasare şi rotire la dreapta: (a) Deplasare logică la dreapta; (b) Deplasare aritmetică la dreapta; (c) Rotire la dreapta.

În cazul unei *deplasări logice*, în poziția rămasă liberă după deplasare se introduce 0, bitul din celălalt capăt fiind pierdut. Aceste deplasări se utilizează în primul rând pentru izolarea unor câmpuri din cadrul unui cuvânt. Biții care nu sunt necesari sunt deplasați în afara cuvântului.

Operația de *deplasare aritmetică* tratează data ca un număr cu semn şi nu modifică bitul de semn al numărului. La deplasarea aritmetică la dreapta, se repetă semnul numărului în poziția rămasă liberă după deplasare. Deplasările aritmetice pot creşte viteza anumitor operații pentru numere cu semn. Dacă numerele sunt reprezentate în complement față de 2 (C2), o deplasare la stânga sau la dreapta cu o poziție corespunde

înmulțirii cu 2, respectiv împărțirii cu 2, cu condiția să nu apară depășire. Pentru numerele fără semn, se pot utiliza şi deplasările logice în locul celor aritmetice.

Operațiile de *rotire* (sau deplasare ciclică) păstrează toți biții operandului. O utilizare posibilă a acestor operații este de a transfera succesiv biții operandului pe poziția cea mai semnificativă, şi de a determina valoarea bitului prin testarea semnului numărului.

## 5.4.4. Instrucțiuni de salt şi apel

Aceste instrucțiuni au rolul de a modifica secvența normală de execuție a instrucțiunilor, operația executată de UCP fiind cea de actualizare a contorului de program cu adresa unei anumite instrucțiuni din memorie.

### 5.4.4.1. Instrucțiuni de salt

O instrucțiune de salt, numită şi de ramificație, are ca un operand adresa următoarei instrucțiuni care trebuie executată. Salturile pot fi *necondiționate* şi *condiționate*. În cazul celor condiționate, contorul de program este actualizat cu operandul instrucțiunii numai dacă este îndeplinită o anumită condiție. În caz contrar, contorul de program este incrementat ca de obicei.

Există două moduri de a genera condiția care va fi testată de o instrucțiune de salt condiționat. În primul rând, în urma execuției unor instrucțiuni aritmetice sau logice vor fi setați indicatorii de condiții ai UCP, în funcție de rezultatul operației. De exemplu, dacă există indicatorii *zero*, *pozitiv*, *negativ* şi *depăşire*, pot exista următoarele instrucțiuni de salt condiționat:

```
BRZ     X       Salt la adresa X dacă rezultatul este zero
BRP     X       Salt la adresa X dacă rezultatul e pozitiv
BRN     X       Salt la adresa X dacă rezultatul e negativ
BRO     X       Salt la adresa X dacă a apărut o depăşire
```

În al doilea rând, se pot executa instrucțiuni de comparare a doi operanzi, care vor seta indicatorii de condiții în funcție de rezultatul comparației (*egal*, *neegal*, *mai mare*, *mai mic*). De exemplu, pot exista instrucțiuni de forma:

```
BRE     X       Salt la adresa X dacă operanzii sunt egali
BRG     X       Salt la adresa X dacă primul operand este mai mare
BRL     X       Salt la adresa X dacă primul operand este mai mic
```

Dacă se utilizează un format cu trei adrese pe instrucțiune, se poate executa o *comparație şi un salt printr-o singură instrucțiune*. De exemplu:

```
BRE     R1,R2,X     Salt la adresa X dacă R1 este egal cu R2
```

Figura 5.4 prezintă exemple de instrucțiuni de salt.

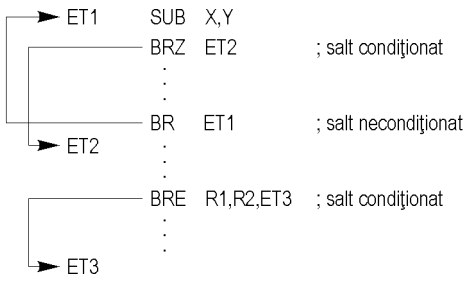


**Figura 5.4.** Instrucțiuni de salt.

Saltul se poate executa la o adresă mai mare decât cea a instrucțiunii curente (înainte), sau la o adresă mai mică (înapoi). Se observă modul în care se poate utiliza un salt necondiționat şi unul condiționat pentru a crea un *ciclu de instrucțiuni* (*buclă*).

### 5.4.4.2. Instrucțiuni de apel

Acestea se utilizează pentru apelul unor *subrutine*, numite şi *proceduri* sau *funcții*. O subrutină este formată dintr-un grup de instrucțiuni, care pot fi apelate din orice punct al programului. După execuția acesteia, se revine la punctul de la care s-a efectuat apelul.

Un avantaj al subrutinelor este că permit reducerea memoriei necesare pentru program. Acelaşi fragment de program poate fi utilizat de mai multe ori. Un alt avantaj este că se permite împărțirea programelor mari în porțiuni mai mici. Prin această *modularizare* se uşurează activitatea de programare.

Pentru utilizarea subrutinelor sunt necesare două instrucțiuni de bază: o instrucțiune *de apel*, care efectuează saltul de la adresa curentă la subrutină, şi o instrucțiune *de revenire*, care permite continuarea programului din punctul de la care s-a efectuat apelul.

Figura 5.5 prezintă execuția subrutinelor în cadrul unui program.

Există un program principal, care începe de la adresa `4000h`. Acest program conține o instrucțiune de apel la subrutina SUB1, care începe de la adresa `4500h`. La întâlnirea acestei instrucțiuni, UCP suspendă execuția programului principal şi începe execuția subrutinei, următoarea instrucțiune fiind încărcată de la adresa `4500h`. În cadrul subrutinei SUB1, există două apeluri la subrutina SUB2, care se află la adresa `4800h`. În ambele cazuri, execuția SUB1 este suspendată şi se trece la execuția SUB2. Instrucțiunea de revenire determină continuarea execuției programului din care s-a efectuat apelul (program apelant), de la instrucțiunea următoare celei de apel.

Deoarece o subrutină poate fi apelată din orice punct al programului, UCP trebuie să memoreze adresa la care trebuie să revină după terminarea subrutinei. Există trei posibilități pentru memorarea acestei adrese de revenire:

- Într-un registru;
- La începutul subrutinei;

- În vârful stivei.

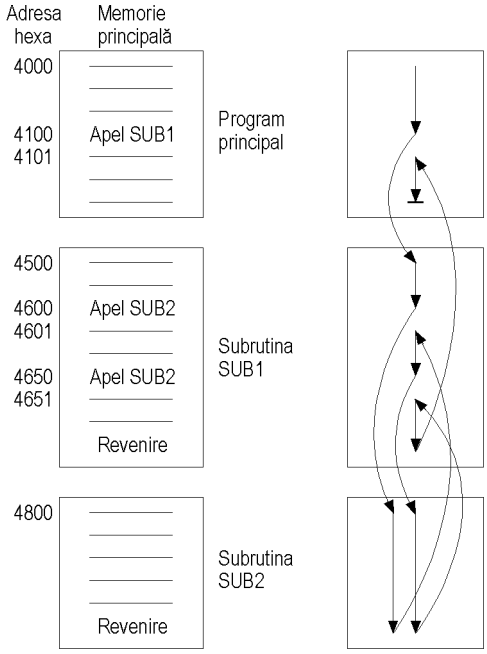


**Figura 5.5.** Execuția subrutinelor dintr-un program.

Se consideră o instrucțiune `CALL X`, care apelează o subrutină aflată la adresa X. Dacă adresa de revenire se salvează într-un registru, la întâlnirea acestei instrucțiuni se execută următoarele operații:

RS ← PC
PC ← X

unde RS este un registru dedicat pentru salvarea adresei de revenire. De menționat că în momentul apelului, PC conține deja adresa următoarei instrucțiuni (cea de după apel).

Dacă adresa de revenire se salvează la începutul subrutinei, operațiile executate sunt:

X ← PC
PC ← PC + 1

Cea mai utilizată metodă este utilizarea stivei. La execuția unui apel, adresa de revenire este depusă în vârful stivei. La terminarea subrutinei, se reface PC cu adresa din vârful stivei. Figura 5.6 ilustrează utilizarea stivei.

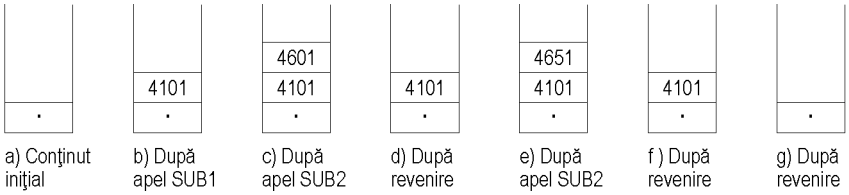


**Figura 5.6.** Utilizarea stivei pentru păstrarea adreselor de revenire ale subrutinelor din Figura 5.5.

La apelul unei subrutine, trebuie să i se transmită acesteia anumiți *parametri*. Aceştia pot fi plasați în *registre* înainte de apel, pot fi depuşi într-o *zonă de memorie*, sau pot fi transmişi prin *stivă*. Ultima metodă este cea mai utilizată de compilatoarele limbajelor de nivel înalt. În acest caz, pe lângă adresa de revenire, se depun în stivă şi parametrii transmişi subrutinei.

### 5.4.5. Instrucțiuni de control al sistemului

Acestea sunt de obicei instrucțiuni *privilegiate*, care pot fi executate numai dacă UCP este într-un mod privilegiat sau execută un program aflat într-o zonă specială de memorie. De obicei, aceste instrucțiuni sunt rezervate sistemului de operare.

Instrucțiunile de control pot citi sau înscrie anumite registre speciale, numite registre de control, sau pot modifica o cheie de protecție a memoriei.

### 5.4.6. Instrucțiuni de I/E

Aceste instrucțiuni permit transferul datelor între un periferic şi UCP. Comunicația cu perifericele se realizează prin registre ale interfețelor de I/E. Aceste registre pot fi:

- Registre de *stare*, care permit citirea stării perifericului;
- Registre de *comandă*, prin care se pot transmite diferite comenzi perifericului;
- Registre de *date*, prin intermediul cărora sunt transferate datele între UCP şi periferic.

## 5.5. Moduri de adresare

Modurile de adresare indică felul în care se specifică operanzii unei instrucțiuni. Aceste moduri de adresare influențează numărul referințelor la memorie pentru obținerea operanzilor, complexitatea calculului adreselor şi flexibilitatea operațiilor care pot fi executate.

Se utilizează următoarele *notații*:

A Conținutul unui câmp de adresă al instrucțiunii;

AE *Adresa efectivă* a locației sau a registrului care conține operandul la are se face referire;

(X) Conținutul locației X.

Fiecare calculator permite mai multe moduri de adresare. Diferite coduri de instrucțiuni pot utiliza diferite moduri de adresare. În cadrul formatului instrucțiunii, există de obicei un *câmp al modului de adresare*, care indică modul de adresare utilizat.

## 5.5.1. Adresarea imediată

Este cea mai simplă formă de adresare, în care operandul se află în cadrul instrucțiunii, în câmpul de adresă:

OPERAND = A

Acest mod poate fi utilizat pentru definirea şi utilizarea constantelor, sau setarea valorii inițiale a variabilelor (Figura 5.7). CO reprezintă codul operației.

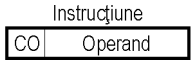


**Figura 5.7.** Adresarea imediată.

Avantajul adresării imediate este că pentru obținerea operandului nu sunt necesare referiri suplimentare la memorie, pe lângă încărcarea instrucțiunii din memorie. Dezavantajul este că dimensiunea operandului este limitată la cea a câmpului de adresă.

## 5.5.2. Adresarea directă

Câmpul de adresă conține adresa efectivă a operandului (Figura 5.8):

AE = A

Acest mod de adresare necesită o singură referire la memorie, şi nu necesită un calcul de adresă. Dezavantajul este că permite un spațiu de adresare limitat, deoarece lungimea câmpului de adresă este de obicei mai mică decât lungimea cuvântului.

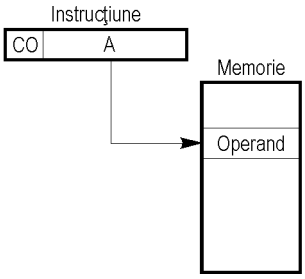


**Figura 5.8.** Adresarea directă.

### 5.5.3. Adresarea indirectă

Pentru a extinde spațiul de adresare, o soluție constă în utilizarea câmpului de adresă din instrucțiune ca o referință la un cuvânt de memorie, care conține adresa completă a operandului. Aceasta reprezintă *adresarea indirectă*:

$$AE = (A)$$

Avantajul este că pentru un cuvânt de lungime $N$, este disponibil un spațiu de adresare de $2^N$. Dezavantajul este că execuția instrucțiunii necesită două referiri la memorie pentru obținerea operandului, una pentru citirea adresei operandului, şi alta pentru valoarea acesteia (Figura 5.9).

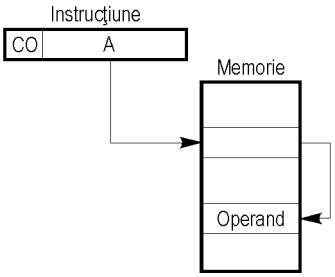


**Figura 5.9.** Adresarea indirectă.

O variantă a adresării indirecte este adresarea indirectă *multinivel*:

$$AE = ( \ldots (A) \ldots)$$

În acest caz, există în cadrul adresei un bit *indicator de indirectare I*. Dacă acesta este 0, adresa reprezintă o adresă efectivă. Dacă este 1, este necesar un alt nivel de indirectare. Dezavantajul este că sunt necesare trei sau mai multe referiri la memorie pentru încărcarea operandului.

### 5.5.4. Adresarea registrelor

Este similară cu adresarea directă. Diferența constă în faptul că adresa se referă la un registru şi nu la memoria principală (Figura 5.10):

AE = R

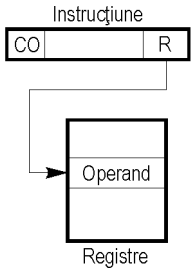


**Figura 5.10.** Adresarea registrelor.

De obicei, un câmp de adresă destinat registrelor are 3 sau 4 biți, astfel că se pot adresa 8 sau 16 registre generale.

Avantajele acestui mod de adresare sunt că este necesar un câmp de adresă de lungime mică, şi nu sunt necesare referiri la memorie pentru citirea operandului. Dezavantajul este că spațiul de adresare este foarte limitat.

### 5.5.5. Adresarea indirectă prin registru

Acest mod de adresare este similar cu adresarea indirectă, dar câmpul de adresă se referă la un registru (Figura 5.11). Astfel,

AE = (R)

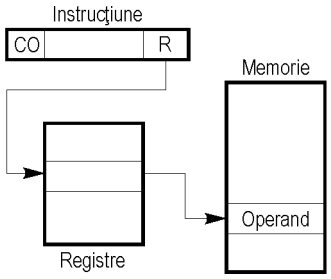


**Figura 5.11.** Adresarea indirectă prin registru.

Avantajele şi limitările adresării indirecte prin registru sunt aceleaşi ca şi pentru adresarea indirectă. Adresarea indirectă prin registru necesită mai puține referiri la memorie decât adresarea indirectă.

### 5.5.6. Adresarea cu deplasament

Acest mod de adresare combină posibilitățile adresării directe cu cele ale adresării indirecte prin registru. Este cunoscut prin diferite denumiri, dar metoda de bază este aceeaşi (Figura 5.12). Adresa efectivă se calculează prin relația:

$$AE = A + (R)$$

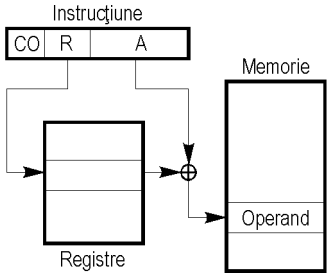


**Figura 5.12.** Adresarea cu deplasament.

Adresarea cu deplasament necesită ca instrucțiunea să aibă două câmpuri de adresă, din care cel puțin unul să fie explicit. Valoarea conținută în primul câmp (A) este utilizată direct. Al doilea câmp de adresă, sau o referință implicită bazată pe codul operației, se referă la un registru al cărui conținut este adunat la A pentru a genera adresa efectivă.

Se vor descrie trei din utilizările cele mai frecvente ale adresării cu deplasament.

#### 5.5.6.1. Adresarea relativă

În acest caz, registrul referit în mod implicit este contorul de program (PC). Deci, adresa instrucțiunii curente este adunată la conținutul câmpului de adresă pentru a genera AE. De obicei, câmpul de adresă este tratat ca un număr în complement față de 2 pentru această operație. Deci, câmpul de adresă este un deplasament relativ la adresa instrucțiunii.

Deoarece majoritatea referințelor la memorie sunt relativ apropiate de instrucțiunea curentă, utilizarea adresării relative reduce numărul de biți de adresă din instrucțiune.

#### 5.5.6.2. Adresarea bazată

Registrul la care se face referire, numit *registru de bază*, conține o adresă de memorie, iar câmpul de adresă conține un deplasament față de acea adresă. Deplasamentul este de obicei un întreg fără semn. Referința la registru poate fi explicită sau implicită.

Acest mod de adresare permite implementarea segmentării memoriei. În anumite implementări, se utilizează un singur registru de bază al segmentului, care este referit în mod implicit. În alte cazuri, se poate selecta un registru care va păstra adresa de bază a unui segment, referirea la acest registru realizându-se în mod explicit.

### 5.5.6.3. Adresarea indexată

Câmpul de adresă conține o adresă de memorie, iar registrul referit, numit *registru de index*, conține un deplasament pozitiv față de acea adresă. Această utilizare este inversă față de cea de la adresarea bazată. Deoarece câmpul de adresă este considerat ca o adresă de memorie, în general conține un număr mai mare de biți decât un câmp de adresă dintr-o instrucțiune cu adresare bazată. Totuşi modul de calcul al adresei efective este acelaşi în ambele cazuri.

O utilizare importantă a indexării este de a asigura un mecanism eficient pentru executarea *operațiilor iterative*. De exemplu, considerăm o listă de numere memorate începând de la adresa A. Presupunem că trebuie să se adune o valoare la fiecare element al listei. Secvența adreselor efective necesare este A, A+1, A+2, …, până la ultima locație a listei. Dacă se utilizează indexarea, adresa A este depusă în câmpul de adresă al instrucțiunii, iar registrul de index este inițializat cu 0. După fiecare operație, registrul de index este incrementat cu 1.

Deoarece registrele de index sunt utilizate în mod frecvent pentru asemenea operații iterative, ele fiind incrementate sau decrementate după fiecare utilizare a lor, anumite sisteme pot executa în mod automat incrementarea sau decrementarea. Această variantă a adresării indexate se numeşte *autoindexare*. Dacă anumite registre sunt dedicate exclusiv indexării, autoindexarea poate fi implicită, fiind executată automat. Dacă se utilizează registre generale, operația de autoindexare trebuie indicată printr-un bit al instrucțiunii.

Autoindexarea prin incrementare poate fi descrisă astfel:

$$AE = A + (R)$$
$$(R) \leftarrow (R) + 1$$

La anumite calculatoare, este posibilă atât adresarea indirectă, cât şi indexarea, şi ele pot fi utilizate în aceeaşi instrucțiune. Există două posibilități: indexarea este executată înainte de indirectare, sau este executată după indirectare.

Dacă indexarea este executată după indirectare, ea se numeşte *postindexare*.

$$AE = (A) + (R)$$

Întâi, conținutul câmpului de adresă este utilizat pentru accesul la o locație de memorie care conține adresa operandului (o adresă directă). Această adresă este apoi indexată prin conținutul registrului.

Această tehnică este utilă pentru accesul la conținutul unui bloc de date cu o structură fixă. Registrul de index va conține deplasamentul în cadrul blocului.

În cazul *preindexării*, indexarea este executată înainte de indirectare:

$$AE = (A + (R))$$

Câmpul de adresă este indexat întâi cu conținutul registrului. Adresa rezultată este utilizată pentru accesul la locația de memorie care conține adresa operandului.

Un exemplu de utilizare a acestei tehnici este pentru construirea unei *tabele de salt.* Într-un anumit punct al programului, poate exista un salt la un număr de locații, în funcție de o anumită condiție. Se poate construi o tabelă cu toate adresele de salt, adresa de început a tabelei fiind indicată în câmpul de adresă al instrucțiunii. Prin indexare în cadrul acestei tabele, se poate găsi adresa de salt necesară.

### 5.5.7. Adresarea stivei

Stiva este implementată de obicei în memoria internă, într-o zonă rezervată de program. Adresa vârfului stivei este conținută într-un registru numit indicator de stivă (SP). Astfel, referirile la locațiile din memoria stivă sunt realizate de fapt utilizând adresarea indirectă prin registru (Figura 5.13).

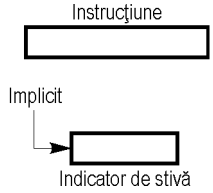


**Figura 5.13.** Adresarea stivei.

Adresarea stivei este o formă a adresării implicite. Instrucțiunile cu stiva nu trebuie să conțină o referință la memorie, operând implicit asupra elementelor din vârful stivei.