

Microprogramare. Proiectare avansată

În capitolul 3 am introdus conceptul microprogramării și am proiectat unitatea de control microprogramată aferentă procesorului didactic. Procesorul didactic are o structură simplă, orientată pe acumulator (vezi 2.1), fapt ce a permis utilizarea formatului (de instrucțiune) cu o singură adresă (vezi 2.2). Microprogramarea se utilizează de regulă la procesoarele CISC, care dispun de un set complex (amplu și variat) de instrucțiuni și care utilizează formatul cu două adrese (vezi 1.3.2).

Prezentul capitol cuprinde un exercițiu de proiectare. Vom stabili arhitectura unui procesor CISC (structura și setul de instrucțiuni) și vom parcurge principalele etape de proiectare a unității de control microprogramate aferentă acestui procesor. Pentru creșterea performanțelor procesorului CISC vom defini și vom utiliza o microinstrucțiune generală, cu format optimizat, în acord cu sugestiile și argumentele expuse pe parcursul paragrafului 3.12.

4.1 Arhitectura procesorului CISC

4.1.1 Schema bloc

Schema bloc aferentă procesorului CISC, a cărui unitate de control urmează să o proiectăm, este redată în figura 4.1. În structura procesorului pot fi identificate următoarele elemente:

SBUS/DBUS/RBUS (busurile interne):

SBUS –*Source Bus* (busul operandului sursă)

DBUS –*Destination Bus* (busul operandului destinație)

RBUS –*Result Bus* (busul rezultatului)

Ieșirile tuturor registrelor din structura procesorului sunt conectate atât la SBUS cât și la DBUS. Prin urmare, conținutul oricărui registru va putea fi emis, fie pe SBUS, prin activarea unei comenzi „*Predă Registrus*”, fie pe DBUS, prin activarea unei comenzi „*Predă Registrud*”. De exemplu, comanda $PdSP_s$ va emite conținutul registrului SP pe SBUS, iar comanda $PdSP_d$ va emite conținutul registrului SP pe DBUS.

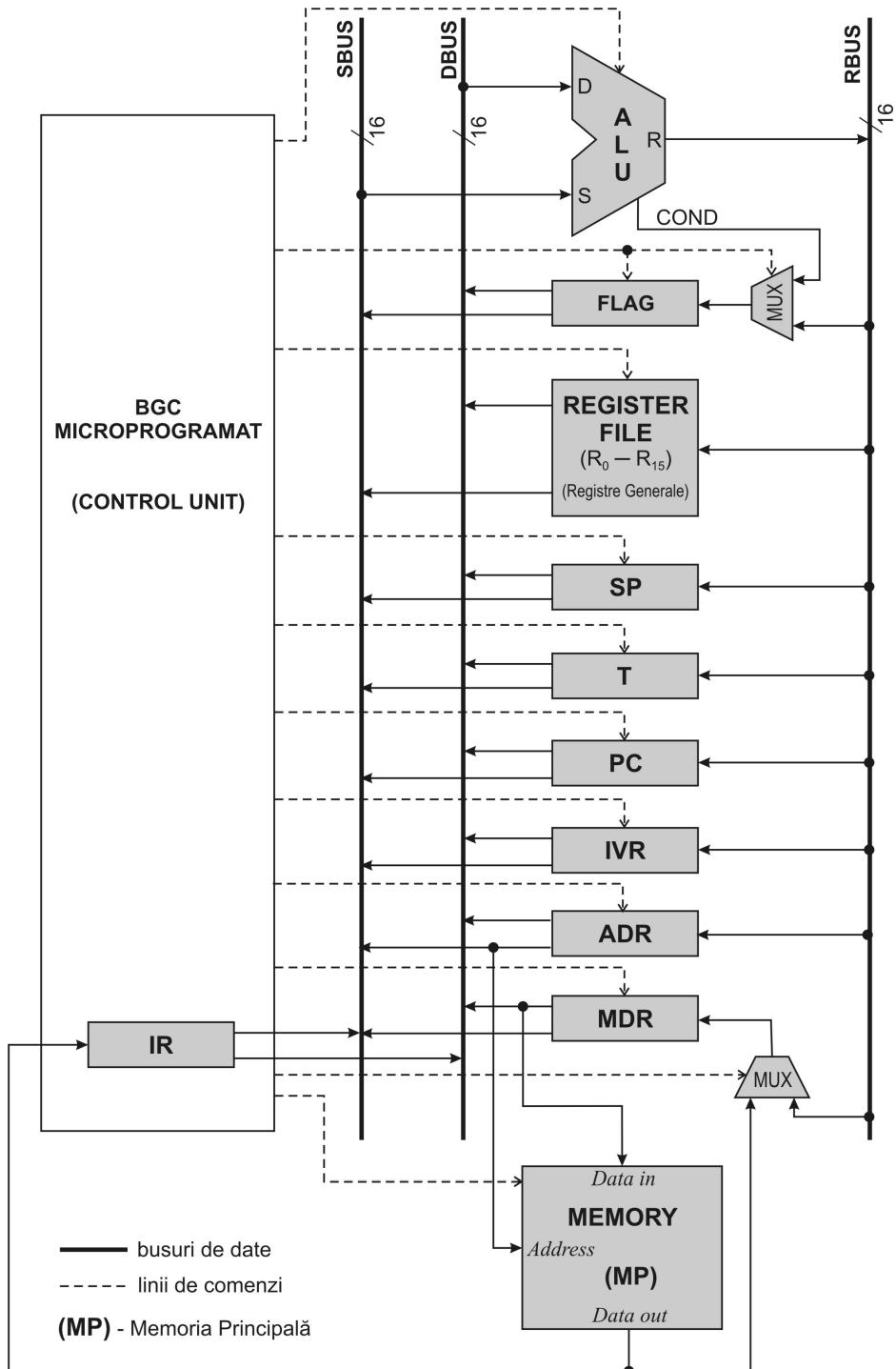


Fig 4.1 Schema bloc aferentă procesorului CISC

ALU –*Arithmetic and Logic Unit*. Este unitatea în care se execută operațiile aritmetico-logice. Unitatea ALU poate prelua doi operanzi (de pe SBUS și respectiv DBUS), efectuează operația aritmetică sau logică dorită și va emite rezultatul pe RBUS. De pe RBUS rezultatul poate fi încărcat în orice registru (mai puțin IR), prin activarea unei comenzi de tip „*Primește Registrul*”. Există evident și operații aritmetico-logice, cum ar fi de exemplu operațiile de deplasare și rotire, care revendică un singur operand. Procesorul CISC nu are o structură orientată pe acumulator și prin urmare, deplasările și rotirile le vom implementa în ALU. Astă înseamnă că, în structura unității ALU vom avea un bloc special de deplasare (SHIFT) de tipul celui propus la 2.7.2.D.

În concluzie, în cadrul fazei (microroutinei) de execuție aferentă instrucțiunilor aritmetice și logice, operanzii (sursă și respectiv destinație) vor fi emisi pe SBUS și respectiv DBUS (prin activarea unor comenzi de tip „*Predă Registrul*”, în ALU (bloc combinațional) se va efectua operația aritmetică sau logică specificată de instrucțiune, rezultatul obținut va fi emis pe RBUS, de unde va putea fi memorat într-un registru (prin activarea unei comenzi de tip „*Primește Registrul*”). Suplimentar, prin activarea unei comenzi PdCOND („*Predă Condiții*”), semnalele de condiții generate de unitatea ALU (C_{out} , Z_R , S_R și DCR) vor fi încărcate în bistabilii de condiții (*C-Carry*, *Z-Zero*, *S-Sign* și *V-oVerflow*) din registrul FLAG.

În structura procesorului pot fi identificate următoarele registre:

FLAG –registru de *flag-uri*. Conține cele 4 *flag-uri* de condiții: C, Z, S și V. Registrul FLAG este foarte popular în arhitecturile (procesoarele) CISC.

La procesoarele CISC, ramificațiile în programe se implementează cu ajutorul unor secvențe de două instrucțiuni, una aritmetică (sau logică), urmată de una de salt condiționat. Prima instrucțiune produce un rezultat și poziționează *flag-urile* de condiții (în acord cu valoarea rezultatului) iar a doua realizează ramificația (condiționată de valoarea rezultatului) prin testarea *flag-ului* aferent.

De exemplu, o ramificație pe condiție de inegalitate poate fi implementată ca în figura 4.2

CMP R1,R2	; R_1-R_2 cu poziționare <i>flag-uri</i> (comparare)
BNE L1	; salt la eticheta L1 dacă $Z=0$
-----	; ramura pe care se continuă (după BNE L1)
-----	; dacă condiția de salt este neândeplinită
L1:	; ramura pe care se continuă (după BNE L1)
.....	; dacă condiția de salt este îndeplinită

Fig 4.2 Utilizarea *flag-urilor* de condiții pentru implementarea ramificațiilor (soluție uzuală la procesoarele CISC)

Instrucțiunea CMP compară R_1 cu R_2 (prin scădere) iar instrucțiunea BNE realizează un salt la eticheta L1 dacă rezultatul comparației (scăderii) este diferit

de 0. Condiția de ramificație va fi de fapt *flag*-ul Z (poziționat mai întâi de instrucțiunea CMP și apoi testat de instrucțiunea BNE).

Dacă la compilarea unui program C, compilatorul alocă registrele R1 și R2 pentru variabilele x și respectiv y, atunci instrucțiunea:

```
if (x==y) {
-----
}
else {
.....
.....
}
```

va genera (prin compilare) exact secvența exemplificată în figura 4.2.

Indicatorii (bistabili) de condiții C, Z, S și V sunt poziționați în fazele (microrutinele) de execuție aferente instrucțiunilor aritmetico-logice astfel încât starea acestor bistabili va caracteriza rezultatul obținut în cadrul instrucțiunii:

C –carry (transport de ieșire din rangul cel mai semnificativ)

C=1 indică rezultat obținut cu transport
C=0 indică rezultat obținut fără transport

Z –zero (rezultat egal cu zero sau diferit de zero obținut pe ieșirile ALU)

Z=1 indică rezultat egal cu zero
Z=0 indică rezultat diferit de zero

S –sign (semn). Caracterizează semnul rezultatului obținut pe ieșirile ALU

S=1 indică rezultat negativ
S=0 indică rezultat pozitiv

V –overflow (depășire)

V=1 indică rezultat incorect (depășirea domeniului de reprezentare)
V=0 indică rezultat corect (fără depășire de domeniu)

Observatie:

Mecanismul de ramificație bazat pe flag-uri de condiții și deci implementat pe două instrucțiuni succesive este evitat la procesoarele RISC, deoarece impune restricții compilatoarelor cu reorganizare. Pentru a minimiza timpul global de procesare a unui program pe o mașină RISC (pentru a maximiza performanțele), compilatorul trebuie să parcurgă două faze succesive: generare cod mașină (translatare), urmată de optimizarea codului mașină generat (reordonarea instrucțiunilor fără a afecta logica programului). Reordonarea are drept scop eliminarea a cât mai multor dependențe de date dintre instrucțiunile programului mașină. Altfel spus, se minimizează numărul de dependențe (hazarduri) de date.

Ramificațiile implementate pe două instrucțiuni succeseive impun restricții compilatorului cu reorganizare. Dacă ne referim la exemplul de mai sus (fig 4.2), între instrucțiunile **CMP** și **BNE** nu pot fi inserate (prin reordonare) instrucțiuni aritmetice sau logice chiar dacă acestea sunt independente (în sensul că nu operează cu regiștrii R1 și respectiv R2), deoarece instrucțiunile aritmetico-logice inserate vor altera condiția de ramificație (vor poziționa flag-urile). Din acest motiv, punctul de ramificație implementat pe două instrucțiuni succeseive este evitat la RISC.

La procesoarele RISC, exceptând instrucțiunile **Load** și **Store**, toate celelalte instrucțiuni operează pe registre, în sensul că pot specifica doar operanzi de tip registru (adresare directă) sau operanzi de tip constantă (adresare imediată); nu pot specifica operanzi în memorie (adresare indirectă sau indexată). Asta înseamnă că toate instrucțiunile aritmetico-logice sunt simplificate (operează exclusiv pe registre) și din acest motiv procesorul RISC se mai numește și mașină load-store.

La RISC, ramificațiile se implementează de regulă pe o singură instrucțiune care, mai întâi efectuează o operație aritmetică-logică (cu operanzi preluati din registre) pentru a evalua condiția testată și apoi realizează saltul dacă condiția testată este îndeplinită. De exemplu, ramificația de tip CISC din figura 4.2 poate fi transformată într-o ramificație echivalentă, de tip RISC (figura 4.3).

Să notăm că optimizarea de cod (la compilare) este utilă și în cazul mașinii CISC. Mașina RISC implică însă un transfer de complexitate din hardware în software și prin urmare, optimizarea codului la compilare este mult mai importantă decât în cazul CISC; performanța mașinii RISC (la execuția programului) este mult mai dependentă de optimizarea realizată la compilare.

BNE R1,R2,L1	; efectuează R ₁ -R ₂ și fă salt la adresa L1 dacă R ₁ -R ₂ ≠0
-----	; ramura pe care se continuă (după BNE R1,R2,L1)
-----	; dacă condiția de salt este neîndeplinită
L1:	; ramura pe care se continuă (după BNE R1,R2,L1)
.....	; dacă condiția de salt este îndeplinită

Fig 4.3 Implementarea ramificațiilor la procesoarele RISC

REGISTER FILE –setul de registre generale. Conține 16 registre pe 16 biți (R₀÷R₁₅), implementate sub forma unei matrici de memorie statică rapidă cu două porturi de citire și un port de scriere (vezi 1.4.2). O astfel de structură multiport permite trei accese concurente în setul de registre:

- un registru (R_i; i=0÷15) citit și emis pe SBUS
- un registru (R_j; j=0÷15) citit și emis pe DBUS
- un registru (R_k; k=0÷15) scris cu datele prezente pe RBUS.

Cele trei accese concurente permit execuția unei operații de tipul $R_k \leftarrow R_i \text{ op } R_j$ pe durata unei singure perioade de tact procesor (op reprezintă orice operație aritmetică sau logică din repertoriul de operații aferent unității ALU).

- PC** –*Program Counter* (denumit uneori *Instruction Pointer*); furnizează adresa instrucțiunii ce urmează a fi extrasă din memorie (*fetch*-ul instrucțiunii). După extragerea instrucțiunii din memorie, registrul PC va fi incrementat (pregătit anticipat pentru adresarea următoarei instrucțiuni).
- IR** –*Instruction Register* (registrul instrucțiunii). În faza (microrutina) **Fetch Instrucțiune**, instrucțiunea selectată în memorie cu adresa existentă în registrul PC, va fi citită și încărcată în IR (*fetch*-ul instrucțiunii). După încărcarea în IR, instrucțiunea va fi decodificată.
- SP** –*Stack Pointer* (registrul pointer de stivă). Stiva este o zonă de memorie rezervată pentru memorări temporare. Instrucțiunea PUSH R_i salvează registrul R_i în stivă iar instrucțiunea POP R_i restaurează registrul R_i din stivă. La PUSH, prin decrementarea registrului SP, o nouă locație se adaugă în vârful stivei, după care data existentă în registrul R_i este copiată în noua locație (salvare R_i). La POP, data existentă în locația din vârful stivei este copiată în R_i (restaurare R_i), după care, prin incrementarea registrului SP, locația citită este eliminată din stivă. Adresarea locațiilor stivei (la instrucțiunile PUSH și respectiv POP) se face deci cu ajutorul registrului SP.
- ADR** –*Address Register* (registrul de adrese). Are rolul de a adresa locațiile de memorie. Când procesorul execută o operație de transfer cu memoria (citire sau scriere), adresa locației de memorie implicată în transferul de date este generată de registrul ADR. Registrul ADR este inaccesibil programatorului (invizibil la nivel *software*).
- MDR** –*Memory Data Register* (registru de date aferent memoriei); furnizează datele de scris în memorie în ciclurile de scriere (conținutul MDR este aplicat pe intrarea “*Data in*” aferentă memoriei principale) și respectiv este încărcat cu datele citite din memorie în ciclurile de citire (ieșirea “*Data Out*” din memorie se aplică prin intermediul MUX 2:1 pe intrarea de date aferentă registrului MDR). Registrul MDR este inaccesibil programatorului (invizibil la nivel *software*).

Prin urmare, o operație de scriere în memorie presupune parcurgerea următoarei secvențe de operații elementare (controlate din microcod):

- încărcarea adresei (locației ce urmează a fi scrisă) în ADR;
- încărcarea datelor de scris în MDR;
- activarea comenzi *WRITE* pentru MP (memoria principală).

O operație de citire din memorie comportă următoarea secvență de operații elementare (controlate din microcod):

- încărcarea adresei (locației ce urmează a fi citită) în ADR;

- activarea comenzi *READ* pentru MP (memoria principală);
- încărcarea datelor (citite din MP) în MDR.

Registrele ARD și MDR sunt deci destinate interfașării cu memoria (pe adrese și respectiv, pe date).

T –registru tampon; utilizat pentru memorări temporare la nivel *hardware*. Este inaccesibil programatorului (invizibil la nivel *software*). Necesitatea registrului T poate fi ilustrată printr-un exemplu. Să considerăm instrucțiunea:

ADD (R1),(R2) ; adună conținutul locației de memorie adresată de R₂
 ; (operandul sursă), cu conținutul locației de memorie
 ; adresată de R₁ (operandul destinație) și depune
 ; rezultatul în locația de memorie adresată de R₁ (peste
 ; operandul destinație, care se va pierde).

Pentru execuția acestei instrucțiuni se vor executa următoarele operații elementare succesive:

- se transferă în ADR conținutul registrului R₂ (adresa operandului sursă);
- se citește operandul sursă din MP (în MDR);
- se transferă operandul sursă din MDR în T (memorare temporară);
- se transferă în ADR conținutul registrului R₁ (adresa operandului destinație);
- se citește operandul destinație din MP (în MDR);
- se adună cei doi operanzi (T+MDR) în ALU și se depune rezultatul în MDR;
- se scrie conținutul MDR (rezultatul) în locația de memorie adresată de ADR (memorarea rezultatului la destinație).

Prin urmare, la instrucțiunile cu 2 operanzi și cu ambii operanzi în memorie (adresare indirectă sau indexată atât la sursă cât și la destinație), sunt necesare două registre tampon (MDR și T) pentru a putea efectua *fetch*-ul celor doi operanzi din memorie.

IVR –*Interrupt Vector Register* (registrul vectorului de intrerupere). În cazul activării unei intreruperi sau excepții, sistemul de intrerupere și excepții va genera vectorul de intrerupere (respectiv de excepție) și-l va încărca în IVR. Vectorul va fi citit din IVR în faza (microrutina) de intrerupere și va fi utilizat pentru localizarea *handler*-ului de intrerupere/excepție, în memorie.

4.1.2 Setul de instrucțiuni

Pentru proiectarea setului de instrucțiuni vom utiliza formatul cu două adrese, format uzuial la procesoarele CISC (vezi 1.3.2) iar pentru localizarea operanzilor vom utiliza patru moduri de adresare (cele mai uzuale): imediat, direct, indirect și indexat. Setul de instrucțiuni aferent procesorului CISC propus este format din patru clase de instrucțiuni:

- instrucțiuni cu doi operanzi
- instrucțiuni cu un singur operand
- instrucțiuni de salt
- instrucțiuni fără operanzi

A. Instrucțiuni cu doi operanzi

Cei doi operanzi sunt denumiți **operand sursă** și respectiv **operand destinație**. Pentru localizarea lor vom utiliza patru moduri de adresare: imediat, direct, indirect și indexat. În funcție de modul de adresare utilizat, operandul (sursă sau destinație) se poate afla într-un registru general sau într-o locație de memorie. Optăm pentru un set de instrucțiuni ortogonal, care va permite orice combinație în ceea ce privește localizarea celor doi operanzi (tabelul 4.1). Cu alte cuvinte, din punctul de vedere al localizării lor, cei doi operanzi ai instrucțiunii se pot regăsi în toate cele patru combinații posibile: registru-registru, registru-memorie, memorie-registru și respectiv memorie-memorie.

Localizare operand sursă	Localizare operand destinație
registrar	registrar
registrar	memorie
memorie	registrar
memorie	memorie

Tabelul 4.1 Cele 4 combinații posibile de localizare a operanzilor la un procesor CISC cu set de instrucțiuni ortogonal

Pentru instrucțiunile cu 2 operanzi vom utiliza formatul cu 2 adrese (fig. 4.4).

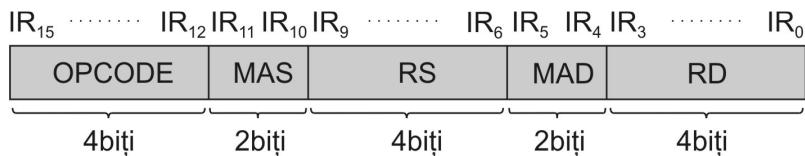


Fig 4.4 Formatul instrucțiunilor cu 2 operanzi

Semnificația câmpurilor evidențiate în formatul instrucțiunii cu 2 operanzi este:

OPCODE	<i>opcode</i> -ul instrucțiunii (codul operației). Este codul propriu-zis al instrucțiunii.
MAS/MAD	–mod de adresare sursă/destinație. Codifică modul de adresare aferent operandului sursă, respectiv destinație.
RS/RD	–registru sursă/destinație. Conține adresa registrului general utilizat pentru localizarea operandului sursă/destinație

Pentru a codifica 4 moduri de adresare sunt necesari 2 biți. Prin urmare, câmpurile MAS și MAD sunt câmpuri de 2 biți. Procesorul conține 16 registre generale (fig 4.1) iar în câmpurile RS și RD se codifică adresa registrului sursă și respectiv adresa registrului destinație. Prin urmare, câmpurile RS și RD sunt câmpuri de 4 biți.

Pe de altă parte, ne propunem proiectarea unui procesor CISC pe 16 biți (fig 4.1). Prin urmare, registrele interne și locațiile de memorie vor fi pe 16 biți. Este firesc să încercăm să codificăm instrucțiunile tot pe 16 biți. În această ipoteză, pentru câmpul OPCODE din figura 4.4 rămân disponibili doar 4 biți dintre care cel puțin un bit (de exemplu cel mai semnificativ) va fi necesar pentru a specifica clasa. Prin urmare, mai rămân doar 3 biți pentru a codifica instrucțiunea în cadrul clasei. Dacă numărul de instrucțiuni din această clasă (clasa instrucțiunilor cu 2 operanzi) nu depășește 8, atunci va fi suficient un OPCODE pe 4 biți (un bit pentru a specifica clasa și 3 biți pentru a codifica instrucțiunile în cadrul clasei). Dacă însă această clasă va conține mai mult de 8 instrucțiuni, atunci câmpul OPCODE va trebui extins la mai mult de 4 biți; drept consecință, instrucțiunile nu mai pot fi codificate pe 16 biți.

Reamintim faptul că, în cazul formatului cu două adrese, adresa rezultatului este identică cu adresa operandului destinație (vezi 1.3.1 și 1.3.2). Prin urmare, rezultatul se depune peste operandul destinație iar prin depunerea rezultatului, operandul destinație se va pierde.

Clasa instrucțiunilor cu 2 operanzi include următoarele instrucțiuni:

- instrucțiuni de transfer:

MOV <i>dest,src</i>	; <i>dest</i> ← <i>src</i>
	; <i>dest</i> =operand destinație
	; <i>src</i> =operand sursă

Exemple:

MOV R0,R1	; R ₀ ←R ₁ (conținutul registrului R ₁ se transferă în registrul R ₀ ; avem adresare directă la ambii operanzi)
-----------	---

MOV R4,(R2)	; R ₄ ←MEM R ₂ (conținutul locației de memorie adresată de R ₂ este transferat în registrul R ₄ . Avem adresare indirectă la sursă și adresare directă la destinație)
-------------	---

ORGANIZAREA ȘI PROIECTAREA CALCULATOARELOR

MOV (R3),124(R5) ; $\text{MEM} | R_3 \leftarrow \text{MEM} | R_5 + 124$ (conținutul locației de
; memorie de la adresa $R_5 + 124$ se transferă în locația
; de memorie adresată de R_3 . Avem adresare indexată
; la sursă și indirectă la destinație; 124 este inexul
; sursă)

- instrucțiuni aritmetice:

ADD dest,src ; $dest \leftarrow dest + src$

SUB dest,src ; $dest \leftarrow dest - src$

CMP dest,src ; $dest - src$ (fără depunerea rezultatului ci doar
; pentru poziționarea flag-urilor de condiții)

Exemple:

ADD (R6),R0 ; $\text{MEM} | R_6 \leftarrow \text{MEM} | R_6 + R_0$ (conținutul locației de
; memorie adresată de R_6 se adună cu conținutul
; registrului R_0 și rezultatul se depune în locația
; de memorie adresată de R_6 . Avem adresare directă
; la sursă și indirectă la destinație)

SUB R3,R5 ; $R_3 \leftarrow R_3 - R_5$ (conținutul registrului R_5 se scade
; din conținutul registrului R_3 și rezultatul
; se depune în R_3 . Avem adresare directă la
; ambii operanzi)

CMP R0,(R1) ; $R_0 - \text{MEM} | R_1$ (conținutul ocației de memorie adresată
; de R_1 se scade din R_0 doar pentru poziționarea
; flag-urilor de condiții conform cu rezultatul scăderii.
; Rezultatul nu se depune la destinație !)

- instrucțiuni logice:

AND dest,src ; $dest \leftarrow dest \text{ AND } src$

OR dest,src ; $dest \leftarrow dest \text{ OR } src$

XOR dest,src ; $dest \leftarrow dest \text{ XOR } src$

Exemple:

AND R2,126 ; $R_2 \leftarrow R_2 \text{ AND } 126$ (operația logică SI între registrul
; R_2 și constanta 124, cu depunerea rezultatului în R_2)

OR R1,(R5) ; $R_1 \leftarrow R_1 \text{ OR } \text{MEM} |_{R_5}$ (operația logică OR între
; locația de memorie adresată de R_5 și registrul R_1 ,
; cu depunerea rezultatului în R_1)

XOR R3,8 ; $R_3 \leftarrow R_3 \text{ XOR } 8$ (operația logică XOR între registrul
; R_3 și constanta 8, cu depunerea rezultatului în R_3)

Am utilizat următoarele notații consacrate:

- R_i specifică conținutul registrului R_i (adresare directă)
- (R_i) specifică conținutul locației de memorie adresată de R_i (adresare indirectă)
- $k(R_i)$ specifică conținutul locației de memorie selectată de adresa R_i+k
(adresare indexată, k reprezentând indexul)
- k specifică operand de tip constantă cu valoarea k (adresare imediată). În cod
mașină, constanta k (codificată în binar) se stochează în memorie, imediat după
codul instrucțiunii (vezi figura 4.9 cu explicațiile aferente).

Instrucțiunile aritmetice și logice poziționează și *flag*-urile de condiții în acord cu rezultatul.
La instrucțiunile aritmetice se poziționează toate *flag*-urile (C, Z, S, V) iar la cele logice se
poziționează doar *flag*-urile Z și S; la instrucțiunile logice nu poate să rezulte niciodată
 $C=1$ sau $V=1$ și prin urmare, nu are sens poziționarea acestor două *flag*-uri.

Făcând inventarul, constatăm că am definit 7 instrucțiuni cu 2 operanzi (MOV, ADD, SUB, CMP, AND, OR și XOR). Pentru a codifica 7 instrucțiuni, vom utiliza cei mai
puțin semnificativi 3 biți din câmpul OPCODE (biții $IR_{14} \div IR_{12}$). Bitul IR_{15} (cel mai
semnificativ) rămâne disponibil pentru a defini clasa:

- $IR_{15}=0$ -specifică instrucțiuni cu 2 operanzi (clasa A)
- $IR_{15}=1$ -specifică instrucțiuni din alte clase

Să observăm că, operandul destinație nu poate fi de tip constantă deoarece, prin
depunerea rezultatului, constanta își modifică valoarea și prin această modificare a valorii,
își pierde în fond statutul de constantă. Prin urmare, adresarea imediată la destinație
rezrezintă un nonsens și de aceea este considerată cod ilegal (instrucțiune ilegală).

B. Instrucțiuni cu un singur operand

Operandul unic utilizat de instrucțiunile din această clasă va avea semnificația de operand
destinație. Aceasta va fi localizat utilizând aceleași patru moduri de adresare (direct, imediat,
indirect și indexat).

Formatul instrucțiunilor cu un singur operand este redat în figura 4.5.

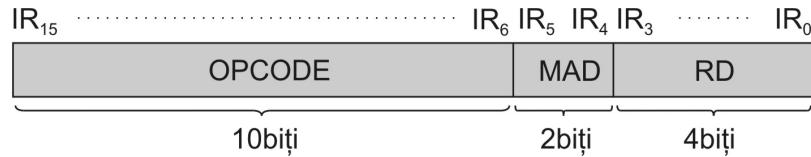


Fig 4.5 Formatul instrucțiunilor cu un singur operand

Semnificația câmpurilor evidențiate în formatul instrucțiunii cu un singur operand este:

OPCODE –opcode-ul instrucțiunii (codul operației)

MAD –mod de adresare destinație. Codifică modul de adresare aferent operandului destinație.

RD –registru destinație (adresa registrului general utilizat pentru localizarea operandului destinație)

Clasa instrucțiunilor cu un singur operand include următoarele instrucțiuni:

- instrucțiuni logice și aritmetice:

CLR dest ; *dest*←0

NEG dest ; *dest*← $\overline{\text{dest}}$

INC dest ; *dest*←*dest*+1

DEC dest ; *dest*←*dest*-1

Exemple:

CLR (R0) ; MEM | R₀←0 (încarcă constanta 0 în locația ; de memorie adresată de R₀)

NEG R3 ; R₃← $\overline{R_3}$ (inversează bit cu bit operandul din R₃)

INC (R2) ; MEM | R₂←MEM | R₂+1 (incrementează cu 1 ; conținutul locației de memorie adresată de R₂)

DEC R5 ; R₅←R₅-1 (decrementează cu 1 conținutul ; registrului R₅)

- instrucțiuni de deplasare și rotire aritmetică și logică:

ASL dest ; deplasare aritmetică la stânga *dest* (*Arithmetic Shift Left*)

ASR <i>dest</i>	; deplasare aritmetică la dreapta <i>dest</i> (<i>Arithmetic Shift Right</i>)
LSR <i>dest</i>	; deplasare logică la dreapta <i>dest</i> (<i>Logical Shift Right</i>)
ROL <i>dest</i>	; rotire la stânga <i>dest</i> (<i>ROtate Left</i>)
ROR <i>dest</i>	; rotire la dreapta <i>dest</i> (<i>ROtate Right</i>)
RLC <i>dest</i>	; rotire la stânga cu <i>carry dest</i> (<i>Rotate Left through Carry</i>)
RRC <i>dest</i>	; rotire la dreapta cu <i>carry dest</i> (<i>Rotate Right through Carry</i>)

Exemple:

ASL R1	; deplasare aritmetică la stânga cu o poziție ; a operandului din R1 (înmulțire cu 2)
ASR (R2)	; deplasare aritmetică la dreapta cu o poziție ; a conținutului locației de memorie adresată ; de R2 (împărțire la 2)
LSR 14(R0)	; deplasare logică la dreapta cu o poziție ; a conținutului locației de memorie de la ; adresa R0+14
RLC R7	; rotire la stânga cu <i>carry</i> a registrului R7

- instrucțiunile PUSH Ri și POP Ri

PUSH Ri	; salvează în stivă registrul Ri
POP Ri	; restaurează din stivă registrul Ri

Exemple:

PUSH R3	; stiva←R3 (se adaugă o locație în vârful ; stivei și în noua locație se memorează R3)
POP R5	; R5←stiva (conținutul locației din vârful stivei ; se transferă în R5 și locația se elimină din stivă)

Similar cazului instrucțiunilor cu doi operanzi, adresarea imediată la destinație reprezintă cod ilegal (instrucțiune ilegală).

Instrucțiunile aritmertice și logice (inclusiv cele de deplasare) poziționează *flagurile* de condiții în conformitate cu rezultatul obținut. Deplasările și rotirile se vor efectua cu ajutorul unui bloc SHIFT, bloc combinațional din compoñența ALU (fig 2.59). Dacă în figura 2.60 înlocuim registrul A cu blocul SHIFT, obținem schemele de deplasare și rotere pe care blocul SHIFT trebuie să le efectueze iar dacă în figura 2.62 înlocuim registrul A cu blocul SHIFT obținem acțiunea instrucțiunilor de deplasare și rotere asupra intrărilor extreme aferente blocului SHIFT (intrarea din extrema dreaptă și respectiv intrarea din extrema stângă).

Dat fiind faptul că operațiile de deplasare și rotere le vom implementa în ALU (în blocul SHIFT), microcomenzile de deplasare și rotere vor deveni comenzi pentru ALU și deci le vom codifica în microinstrucțiune în cîmpul OPERAȚIE ALU (fig. 4.18)

C. Instrucțiunile de salt

Această clasă conține instrucțiuni de salt condiționat, instrucțiunea de salt necondiționat și instrucțiunea de apel procedură (instrucțiunea CALL).

Instrucțiunile de salt condiționat (instrucțiunile de *branch*) sunt necesare pentru implementarea buclelor de program. Cu ajutorul acestora se implementează, în codul mașină, ramificațiile condiționate (vezi explicațiile relative la registrul FLAG de la 4.1.1). Codul mașină sau formatul executabil al unui program se obține prin compilarea aplicației aferente, scrisă într-un limbaj de nivel înalt. Constructorii de buclă din limbajele de nivel înalt (de exemplu **if**, **while**, **do while** din C), la compilare, sunt translatați în cod mașină utilizând instrucțiuni de ramificație (de salt condiționat).

Construcțiile pe care le regăsim în limbajele de nivel înalt revindică trei tipuri de instrucțiuni de salt:

- **salturi directe.** Sunt salturile în care adresa de salt este specificată sub forma unei constante sau sub forma unei etichete. În limbajul de asamblare se utilizează etichete pentru a scuti programatorul de obligația (complicăția) de a calcula, el însuși, adresele de salt. De exemplu, instrucțiunea JMP ET1 are semnificația „salt la eticheta ET1”. Când programul este translatat în cod mașină, compilatorul (asamblorul) generează o valoare binară concretă pentru eticheta ET1, valoare care va fi integrată în codul instrucțiunii și care va avea semnificația de adresă de salt (JMP *adr*). Salturile directe se implementează în variantă relativă și nu absolută, în sensul că în codul instrucțiunii se codifică de fapt un OFFSET și nu o adresă de salt absolută (vezi explicațiile relative la instrucțiunile de salt aferente procesorului didactic, paragraful 2.7.1.I, figurile 2.42 și 2.43, precum și exemplificarea saltului direct cu *offset* pozitiv din figura 2.44). Salturile relative facilitează relocarea programelor în memorie.
- **salturi indirecte.** Sunt salturile în care adresa de salt este specificată de un registru general. De exemplu, instrucțiunea JMP (R5) are semnificația „salt la adresa specificată de registrul R5”.
- **salturi indexate.** Sunt salturile în care adresa de salt se obține prin însumarea a două componente: registrul specificat + indexul specificat. De exemplu, instrucțiunea JMP 18(R3) are semnificația „salt la adresa R₃+18”.

Prin urmare, instrucțiunile de salt sunt instrucțiuni viabile cu trei moduri de adresare: imediat, indirect și indexat. O instrucțiune de salt cu mod de adresare direct reprezintă cod ilegal.

În altă ordine de idei, instrucțiunile de salt sunt de fapt instrucțiuni cu un singur operand în care operandul are o semnificație specială; operandul unei instrucțiuni de salt este de fapt adresa de salt. Prin urmare, formatul instrucțiunilor de salt va fi cel din figura 4.5 (viabil cu modurile de adresare AM, AI, AX și respectiv cod ilegal cu AD), cu precizarea că **operandul instrucțiunii va fi interpretat ca adresă de salt**.

Salturile directe revendică codificarea *offset*-ului și integrarea acestuia în codul instrucțiunii. Dacă la procesorul didactic *offset*-ul a fost codificat pe 8 biți (figura 2.43), la procesorul CISC pe care-l descriem, *offset*-ul va fi codificat pe 16 biți și va fi stocat în memorie imediat după instrucțiunea de salt de care ține. În figura 4.6.a) este redat modul de codificare (formatul) unei asemenea instrucțiuni de salt. Dacă codul instrucțiunii este stocat în memorie la adresa (generică) PC, atunci *offset*-ul va fi stocat la adresa următoare (PC+2) iar următoarea instrucțiune a programului la adresa PC+4. De fapt, codul instrucțiunii de salt va avea în acest caz lungimea de 32 de biți (două cuvinte succesive de memorie) deoarece va îngloba și *offset*-ul. Iată cum la CISC, se ajunge la formatul de lungime variabilă.

Salturile indexate revendică de asemenea codificarea unei constante în codul instrucțiunii. Constanta va fi interpretată ca fiind index și nu *offset*. Codul unei asemenea instrucțiuni de salt se întinde tot pe 32 de biți (două cuvinte succesive de memorie) și este redat în figura 4.6.b).

Salturile indirekte nu utilizează constante; pentru generarea adresei de salt acestea utilizează doar un registru general. Prin urmare, codul unei asemenea instrucțiuni rămâne pe 16 biți și este redat în figura 4.6.c).

Semnificația câmpurilor evidențiate în formatul instrucțiunilor de salt (fig 4.6) este:

OPCODE –*opcode*-ul instrucțiunii de salt (codul operației)

MAD –mod de adresare destinație. Codifică modul de adresare aferent registrului utilizat în generarea adresei de salt.

RD –registru destinație (adresa registrului general utilizat pentru generarea adresei de salt)

În cazul salturilor directe, codul instrucțiunii include și **OFFSET**-ul. **OFFSET**-ul este un număr cu semn (reprezentat în cod complementar) care indică sensul saltului și numărul de adrese sărit. Dacă **OFFSET**-ul este negativ (dacă bitul de semn are valoarea 1), saltul va fi înapoi, iar dacă **OFFSET**-ul este pozitiv (dacă bitul de semn are valoarea 0), saltul va fi înainte. Saltul este relativ la PC-ul curent. Prin PC_current se înțelege adresa instrucțiunii care succede instrucțiunii de salt (fig 2.44). Prin urmare, pentru a obține adresa de salt, în faza (microrutina) de execuție se va efectua o adunare: se adună **OFFSET**-ul cu valoarea existentă în registrul PC. Rezultatul (adresa de salt) se depune în registrul PC. La salturile condiționate, încărcarea în PC se face numai dacă condiția de salt este îndeplinită. În cazul în care condiția de salt este neîndeplinită, se inhibă încărcarea în PC a adresei de salt și prin urmare, programul va continua secvențial.

În cazul salturilor indexate, codul instrucțiunii include și **INDEX**-ul. **INDEX**-ul se adună la conținutul registrului general specificat și astfel se obține adresa de salt.

ORGANIZAREA ȘI PROIECTAREA CALCULATOARELOR

Salturile indirekte nu conțin constante integrate în codul instrucțiunii. Conținutul registrului general specificat reprezintă chiar adresa de salt.

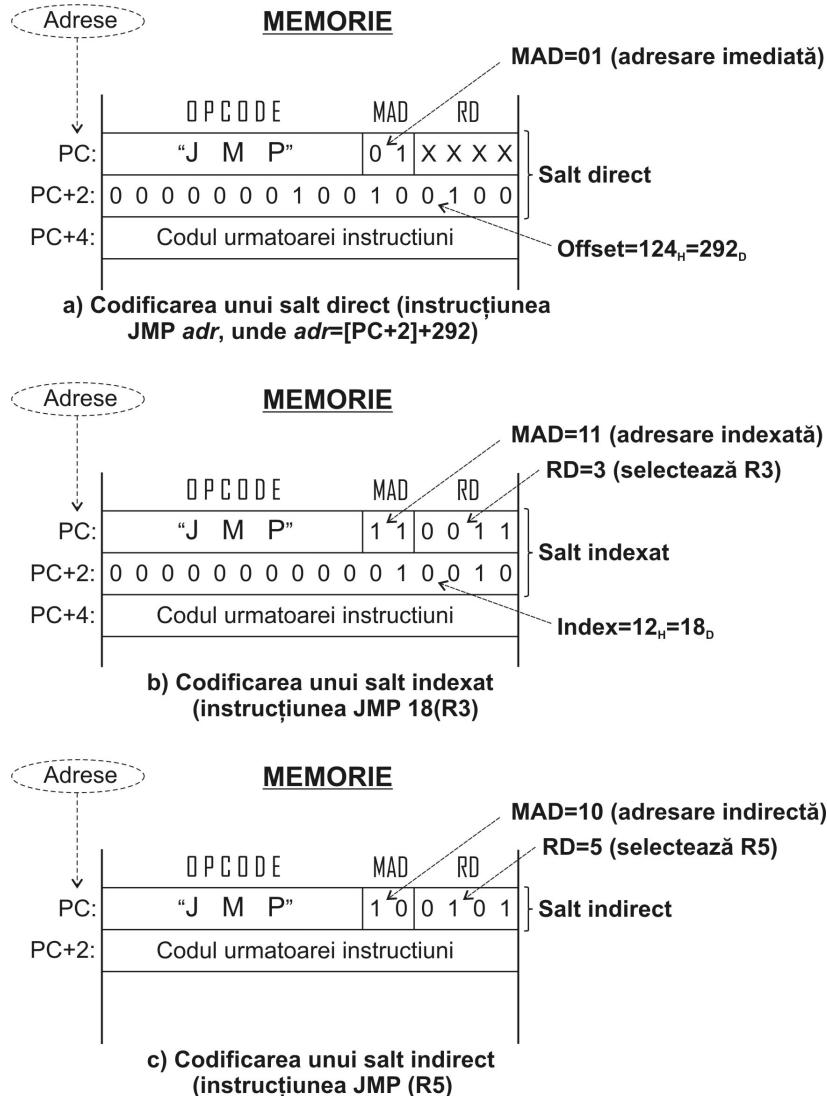


Fig 4.6 Formatul instrucțiunilor salt

Clasa instrucțiunilor de salt condiționat conține următoarele instrucțiuni:

- instrucțiuni de salt condiționat:

BEQ adr ; salt la adresa adr dacă flag-ul Z=1 (*branch if equal*)

BNE adr	; salt la adresa <i>adr</i> dacă <i>flag</i> -ul Z=0 (<i>branch if not equal</i>)
BMI adr	; salt la adresa <i>adr</i> dacă <i>flag</i> -ul S=1 (<i>branch if minus</i>)
BPL adr	; salt la adresa <i>adr</i> dacă <i>flag</i> -ul S=0 (<i>branch if plus</i>)
BCS adr	; salt la adresa <i>adr</i> dacă <i>flag</i> -ul C=1 (<i>branch if carry is set</i>)
BCC adr	; salt la adresa <i>adr</i> dacă <i>flag</i> -ul C=0 (<i>branch if carry is clear</i>)
BVS adr	; salt la adresa <i>adr</i> dacă <i>flag</i> -ul V=1 (<i>branch if overflow is set</i>)
BVC adr	; salt la adresa <i>adr</i> dacă <i>flag</i> -ul V=0 (<i>branch if overflow is clear</i>)

- instrucțiunea de salt necondiționat:

JMP adr ; salt la adresa *adr*

- instrucțiunea de apel procedură (subrutină):

CALL adr ; apelul procedurii de la adresa *adr*

Exemple:

JMP 24(R7) ; salt necondiționat la adresa R₇+24

BEQ (R1) ; dacă Z=1: salt la adresa specificată de R₁
; dacă Z=0: treci sevențial la următoarea instrucțiune

CALL ET1 ; apelul procedurii de la eticheta ET1.

Observatie:

În ideea de a ușura munca programatorului, în limbajul de asamblare, apelul unei proceduri se face de regulă utilizând numele acesteia. De exemplu, instrucțiunea **CALL Proc1** apelează procedura cu numele *Proc1*. Pentru a obține **OFFSET**-ul, asamblorul va calcula diferența:

$$\text{OFFSET} = \text{Adresa_Proc1} - \text{PC_revenire} \quad (4.1)$$

unde:

Adresa_Proc1 reprezintă adresa primei instrucțiuni din procedura *Proc1*

PC_revenire reprezintă adresa instrucțiunii care succede instrucțiunii **CALL**

D. Instrucțiuni fără operanzi

În această clasă includem instrucțiunile care nu referă (în manieră explicită) operanzi și care, pe cale de consecință, nu pot fi încadrate în clasele anterioare. Mai exact, clasa conține instrucțiuni fără operanzi și instrucțiuni cu operanzi referiți implicit (prin **OPCODE**) și nu explicit prin câmpuri de adresă registru (**RS**, **RD**) și respectiv mod de adresare (**MAS**, **MAD**) codificate în formatul instrucțiunii. Concret, clasa va conține: instrucțiuni de control, instrucțiuni de poziționare a *flag*-urilor de condiții, instrucțiuni de revenire din procedură și respectiv din *handler*-ul de intrerupere, etc.

Formatul instrucțiunilor fără operanzi este redat în figura 4.7.

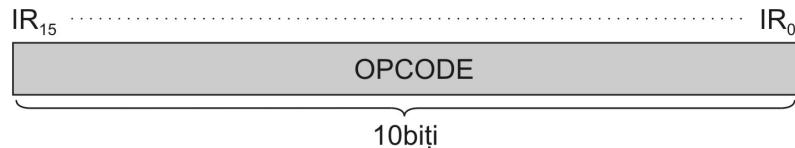


Fig 4.7 Formatul instrucțiunilor fără operanzi

Întreg codul de 16 biți este alocat câmpului **OPCODE** care reprezintă codul operației.

Clasa instrucțiunilor fără operanzi include următoarele instrucțiuni:

- instrucțiuni de poziționare a *flag*-urilor de condiții:

CLC	; C←0 (<i>CLear Carry</i>)
CLZ	; Z←0 (<i>CLear Zero</i>)
CLS	; S←0 (<i>CLear Sign</i>)
CLV	; V←0 (<i>CLear oVerflow</i>)
CCC	; C/V/Z/S←0 (<i>Clear Condition Code</i>)
SEC	; C←1 (<i>SEt Carry</i>)
SEZ	; Z←1 (<i>SEt Zero</i>)
SES	; S←1 (<i>SEt Sign</i>)
SEV	; V←1 (<i>SEt oVerflow</i>)
SCC	; C/V/Z/S←1 (<i>Set Condition Code</i>)

- instrucțiuni de control, de revenire din procedură și respectiv din *handler*-ul de întrerupere și de salvare/restaurare în/din stivă a registrelor PC și respectiv FLAG:

NOP	; nici o operație (<i>No OPeration</i>)
HALT	; oprire
EI	; validare întreruperi (<i>Enable Interrupts</i>)
DI	; invalidare întreruperi (<i>Disable Interrupts</i>)
PUSH PC	; stiva←PC
POP PC	; PC←conținutul locației din vârful stivei
PUSH FLAG	; stiva←FLAG (registrul de <i>flag</i> -uri)
POP FLAG	; FLAG←conținutul locației din vârful stivei
RET	; revenire din procedură (<i>RETurn</i>)
IRET	; revenire din întrerupere (<i>Interrupt RETurn</i>)

4.1.3 Modurile de adresare

În cazul instrucțiunilor cu 2 operanzi, localizarea operandului sursă se face cu ajutorul câmpurilor MAS și respectiv RS iar localizarea operandului destinație se face cu ajutorul câmpurilor MAD și respectiv RD (fig. 4.4). În cazul instrucțiunilor cu un singur operand, operandul unic este întotdeauna operand destinație și localizarea acestuia se face cu ajutorul câmpurilor MAD și respectiv RD (fig. 4.5).

Câmpurile MAS (biții IR₁₁ și IR₁₀) și MAD (biții IR₅ și IR₄) sunt câmpuri de doi biți, care permit codificarea a patru moduri de adresare (tabelele 4.2 și 4.3)

MAS		Mod de adresare sursă
IR ₁₁	IR ₁₀	
0	0	Direct (AD)
0	1	Imediat (AM)
1	0	Indirect (AI)
1	1	Indexat (AX)

Tabelul 4.2 Codificarea modurilor de adresare pentru operandul sursă

MAD		Mod de adresare destinație
IR ₅	IR ₄	
0	0	Direct (AD)
0	1	Imediat (AM)
1	0	Indirect (AI)
1	1	Indexat (AX)

Tabelul 4.3 Codificarea modurilor de adresare pentru operandul destinație

Reamintim faptul că adresarea imediată la destinație reprezintă cod ilegal deoarece ar presupune memorarea rezultatului într-o constantă (nonsens).

În cele ce urmează vom explica modul de localizare a operandului, în cadrul celor patru moduri de adresare, pe baza unor scheme de principiu:

A. Modul de adresare direct

Selecția unui operand aflat într-un registru general se poate face numai cu ajutorul modului de adresare direct. Dacă luăm în considerare o instrucțiune cu doi operanzi (fig. 4.4), cu adresare directă la ambii operanzi (sursă și respectiv destinație), atunci operandul sursă se va afla în registru general selectat de adresa codificată binar în câmpul RS iar operandul destinație se va afla în registru general selectat de adresa codificată binar în câmpul RD. În mod uzual, registru general selectat de câmpul RS se numește **registru sursă** iar registru selectat de câmpul RD se numește **registru destinație**. Prin urmare, în cazul modului de adresare direct (atât la sursă cât și la destinație), operandul sursă se află în registru sursă iar operandul destinație se află în registru destinație.

Fie instrucțiunea de adunare:

ADD R2,R8 ; R₂←R₂+R₈

Această instrucțiune adună registru R₂ (operandul destinație) cu registru R₈ (operandul sursă) și depune rezultatul în registru R₂. Prin urmare, instrucțiunea utilizează adresare directă atât la sursă cât și la destinație. În figura 4.8 este evidențiat codul mașină al instrucțiunii precum și modul de localizare a celor doi operanzi. Valorile câmpurilor din codul (din formatul) acestei instrucțiuni sunt:

OPCODE=“ADD” - o combinație binară care reprezintă *opcode*-ul instrucțiunii ADD

MAS=00 -adresare directă la sursă (tabelul 4.2)

RS=1000_B=8_D -operandul sursă este R₈ (conținutul registru R₈)

MAD=00 -adresare directă la destinație (tabelul 4.3)

RD=0010_B=2_D -operandul destinație este R₂ (conținutul registru R₂)

Observație:

Unitatea adresabilă în memorie este octetul, iar codul instrucțiunii este pe 16 biți (2 octeți). Dacă instrucțiunea ADD R2,R8 se află în memorie la adresa (generică) PC, atunci următoarea instrucțiune se va afla la adresa PC+2.

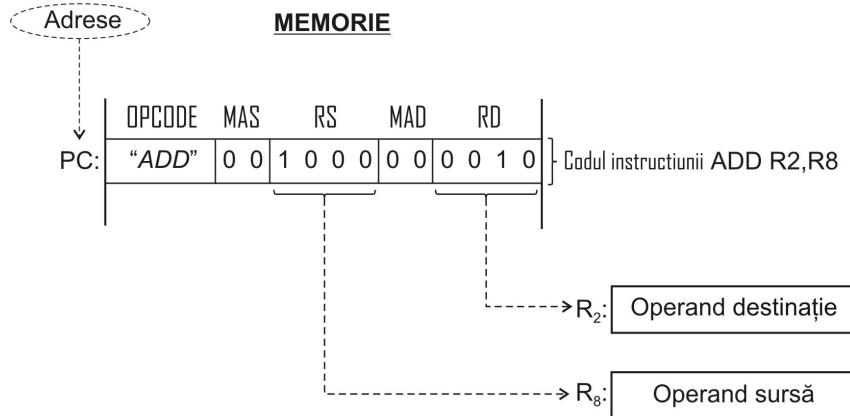


Fig 4.8 Localizarea operanzilor sursă și destinație în cazul adresării directe

B. Modul de adresare imediat

Adresarea imediată este utilizată pentru specificarea operanzilor de tip constantă. După cum am precizat la 4.1.2.A și respectiv 4.1.2.B, adresarea imediată este legală la sursă și ilegală la destinație.

Fie instrucțiunea:

MOV R0,124H ; R₀←124H („move”)

Operandul sursă al acestei instrucțiuni este constanta $124_{\text{H}}=0000000100100100_2$ (adresare imediată) iar operandul destinație este registrul R₀ (adresare directă). Operandul imediat va fi stocat în memoria în imediata vecinătate a codului instrucțiunii (fig. 4.9). Dacă instrucțiunea se află în memorie, la adresa (generică) PC, atunci operandul imediat va fi memorat la adresa PC+2 (în următoarea locație de memorie) iar codul următoarei instrucțiuni (din cadrul programului) va fi memorat la adresa PC+4. În realitate, codul instrucțiunii MOV R0,124H înglobează și operandul imediat, va ocupa două cuvinte succesive în memorie și va avea deci lungime dublă ($2 \times 16 = 32$ de biți). Iată cum la CISC se ajunge la formatul de lungime variabilă; lungimea variabilă a instrucțiunii CISC nu este un deziderat al proiectantului ci o consecință inevitabilă și nedorită.

În final, să evidențiem valorile câmpurilor din codul (din formatul) instrucțiunii exemplificate:

OPCODE=“MOV” - o combinație binară care reprezintă *opcode*-ul instrucțiunii MOV

- | | |
|--------------------------------------|---|
| MAS=01 | -adresare imediată la sursă (tabelul 4.2) |
| RS=XXXX _B | -în cazul adresării immediate, câmpul RS nu este utilizat |
| MAD=00 | -adresare directă la destinație (tabelul 4.3) |
| RD=0000 _B =0 _D | -operandul destinație este R0 (conținutul registrului R0) |

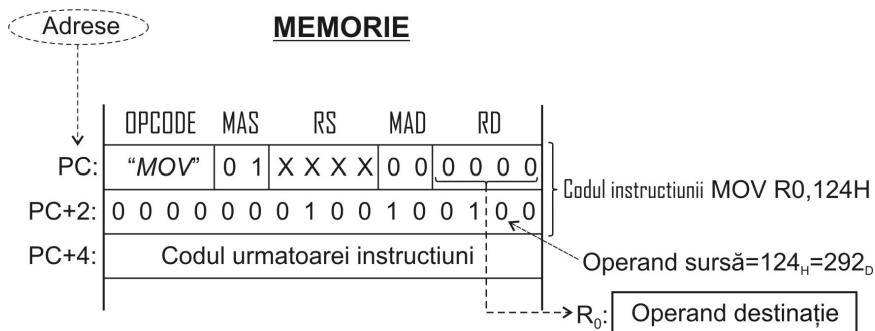


Fig 4.9 Localizarea operandului sursă în cazul modului de adresare imediat

C. Modul de adresare indirect

Modul de adresare indirect este utilizat pentru adresarea operanziilor aflați în memorie. Dacă la adresarea directă operandul se află în registru, la adresarea indirectă operandul se află în memorie la adresa specificată de registru. Prin urmare, la adresarea directă, în registru găsim direct operandul iar la adresarea indirectă, în registru găsim adresa operandului (fig 4.10); pentru obținerea operandului este necesară o operație de citire din memorie de la adresa specificată de registru (indirectare prin registru).

În limbajele de asamblare sau adoptat următoarele convenții:

- Notația R_k se utilizează pentru adresarea directă. Conținutul registrului R_k reprezintă chiar operandul.
- Notația (R_k) se utilizează pentru adresarea indirectă. Registrul R_k conține adresa operandului. Operandul se află în locația de memorie selectată de adresa existentă în registrul R_k .

Fie instrucțiunea:

ADD (R7),(R9) ; MEM |_{R7}←MEM |_{R7+MEM} |_{R9}

Această instrucțiune adună conținutul locației de memorie adresată de R_7 (operandul destinație) cu conținutul locației de memorie adresată de R_9 (operandul sursă) și depune

rezultatul în locația de memorie adresată de R₇. Prin urmare, instrucțiunea utilizează adresare indirectă, atât la sursă cât și la destinație. Codul mașină al instrucțiunii precum și schema care evidențiază modul de localizare a celor doi operanzi sunt prezentate în figura 4.10.

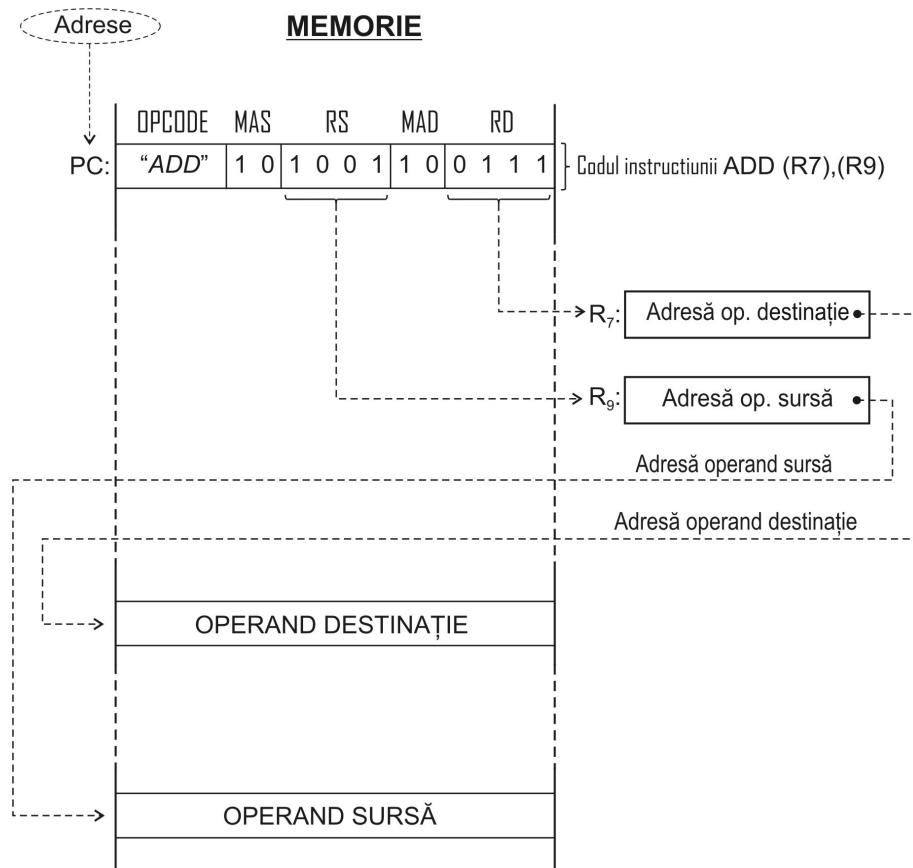


Fig 4.10 Localizarea operanzilor sursă și destinație în cazul adresării indirecte

Valorile câmpurilor din codul (formatul) instrucțiunii exemplificate sunt:

OPCODE=“ADD” -o combinație binară care reprezintă *opcode*-ul instrucțiunii ADD

MAS=10 -adresare indirectă la sursă (tabelul 4.2)

RS=1001_B=9_D -registrul R9 conține adresa operandului sursă

MAD=10 -adresare indirectă la destinație (tabelul 4.3)

RD=0111_B=7_D -registrul R7 conține adresa operandului destinație

D. Modul de adresare indexat

Modul de adresare indexat, ca și modul indirect, este utilizat pentru adresarea operanzilor aflați în memorie. După cum am vazut deja, la adresarea indirectă, adresa operandului se obține din registrul specificat (indirectare prin registru). La adresarea indexată, adresa operandului sursă (destinație) se obține printr-o operație de adunare. Se adună la registrul sursă (registrul destinație) indexul sursă (indexul destinație) codificat binar pe 16 biți și plasat în memorie imediat după codul instrucțiunii. Dacă instrucțiunea are adresare indexată atât la sursă cât și la destinație (ca în figura 4.11), atunci indexul sursă va fi plasat imediat după codul instrucțiunii (la adresa PC+2) iar indexul destinație va fi plasat la adresa PC+4. La adresa PC+6 se va găsi codul următoarei instrucțiuni. Dacă instrucțiunea are adresare indexată doar la unul dintre operanzi (fie sursă, fie destinație), atunci indexul (sursă sau destinație) va fi plasat imediat după codul instrucțiunii (la adresa PC+2); la adresa PC+4 va fi codul următoarei instrucțiuni.

Fie instrucțiunea:

$MOV 142(R3),24(R8), ; MEM |_{R3+142} \leftarrow MEM |_{R8+24}$

Această instrucțiune are adresare indexată, atât la sursă cât și la destinație. În figura 4.11 este prezentat codul mașină și schema de localizare a operanzilor (sursă și respectiv destinație) aferenți acestei instrucțiuni. Valorile câmpurilor din codul (formatul) instrucțiuni exemplificate sunt:

OPCODE=“MOV” -o combinație binară care reprezintă *opcode*-ul instrucțiunii MOV)

MAS=11 -adresare indexată la sursă (tabelul 4.2)

RS= $100_B=8_D$ -registru R8 furnizează o adresă de bază pentru operandul sursă

MAD=11 -adresare indexată la destinație (tabelul 4.3)

RD= $0011_B=3_D$ -registru R3 furnizează o adresă de bază pentru operandul destinație

Operandul sursă al acestei instrucțiuni este reprezentat de conținutul locației de memorie de la adresa $R8+24_D$; R8 este registrul sursă (care furnizează o adresă de bază relativă la operandul sursă) iar $24_D=000000000011000_2$ este indexul sursă. Adresa operandului sursă se obține prin însumarea celor două componente: adresa de bază (preluată din registrul sursă) + index sursă.

Similar, operandul destinație aferent acestei instrucțiuni este reprezentat de conținutul locației de memorie de la adresa $R3+142_D$; R3 este registrul destinație (care furnizează o adresă de bază relativă la operandul destinație) iar $142_D=0000000010001110_2$ este indexul destinație. Adresa operandului se obține prin însumarea celor două componente: adresa de bază (preluată din registrul destinație) + index destinație.

Din figura 4.11 se poate observa că instrucțiunea $MOV 142(R3),24(R8)$ are codul mașină extins la $3 \times 16 = 48$ biți. Pe primii 16 biți s-a codificat codul propriu-zis al instrucțiunii, pe următorii 16 biți s-a codificat indexul sursă iar pe ultimii 16 biți s-a codificat indexul destinație. Putem în sfârșit concluziona că, procesorul nostru CISC

operează cu format de lungime variabilă, având trei tipuri de instrucțiuni (din punctul de vedere al lungimii): pe 16 biți, pe 32 biți și pe 48 biți.

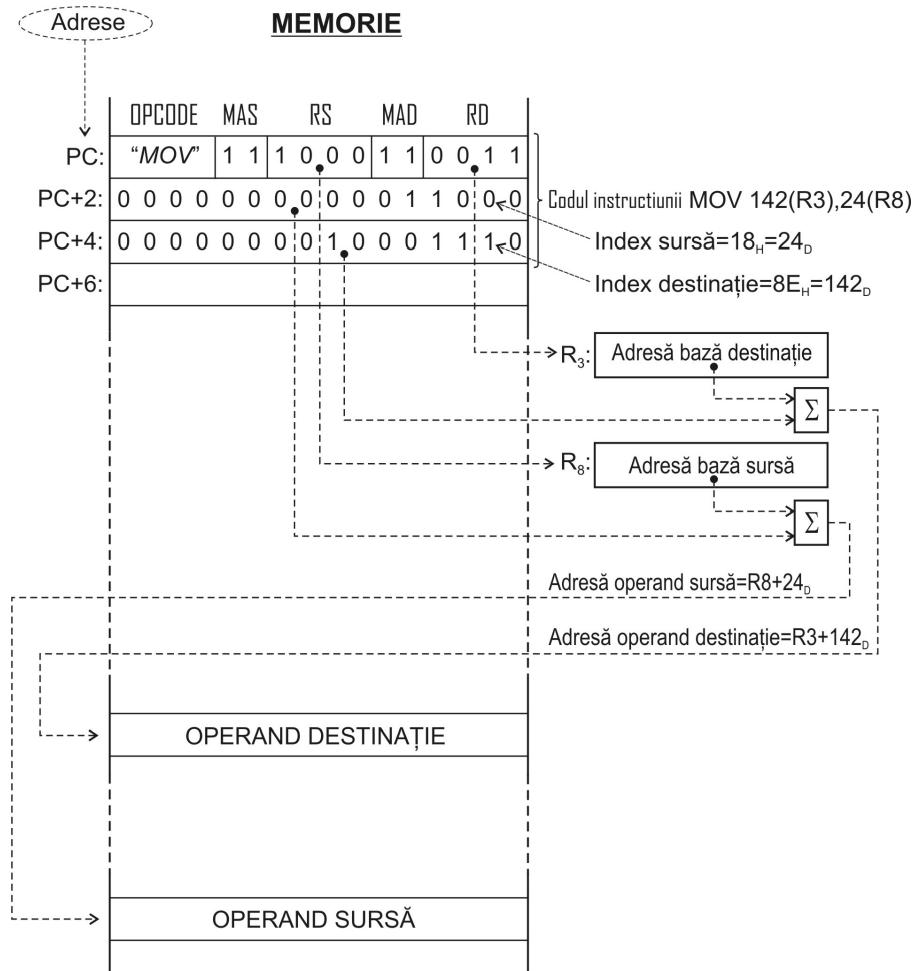


Fig 4.11 Localizarea operanzilor sursă și destinație în cazul adresării indexate

Să mai notăm că, în cazul instrucțiunilor cu doi operanzi, modurile de adresare pot apărea în orice combinație (ortogonalitate). Singura restricție (firescă) rămâne cea menționată dea: adresarea imediată la destinație reprezintă cod ilegal.

În figura 4.12 este prezentat codul mașină și respectiv schema de localizare a operanzilor pentru instrucțiunea **CMP 62(R7),26FH**, instrucțiune ce combină modul de adresare imediat la sursă cu modul de adresare indexat la destinație.

După cum rezultă și din figura 4.12, operandul sursă este constanta 26F_H=623_D iar operandul destinație este conținutul locației de memorie selectată de adresa R₇+62D. Această instrucțiune „compare” compară operandul sursă (*src*) cu operandul destinație

(*dest*), efectuând de fapt operația de scădere *dest-src* și poziționează *flag*-urile de condiții conform cu rezultatul comparației (scăderii).

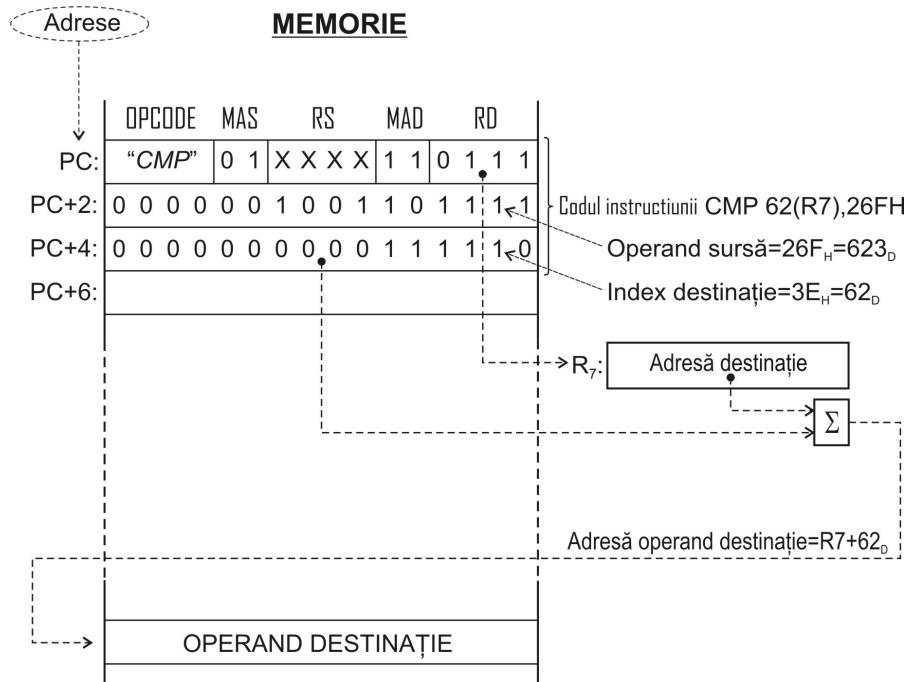


Fig 4.12 Localizarea operanzilor sursă și destinație aferenți instrucțiunii CMP 62(R7),26FH

4.1.4 Codificarea instrucțiunilor

La 4.1.1 a fost definită structura internă aferentă procesorului CISC iar la 4.1.2 a fost definit setul de instrucțiuni aferent acestui procesor. Setul de instrucțiuni cuprinde 4 clase de instrucțiuni (clasele A, B, C și D). Clasa A (instrucțiuni cu 2 operanzi) conține 7 instrucțiuni, clasa B (instrucțiuni cu un singur operand) conține 13 instrucțiuni, clasa C (instrucțiuni de salt) conține 10 instrucțiuni și clasa D (instrucțiuni fără operanzi) conține 20 instrucțiuni (în total 7+13+10+20=50 instrucțiuni).

Trebuie să precizam că, cele 7 instrucțiuni din clasa A, cele 13 instrucțiuni din clasa B și cele 10 instrucțiuni din clasa C sunt instrucțiuni generice. De exemplu, instrucțiunea **MOV dest, src** este o instrucțiune generică care acceptă 4 moduri de adresare la operandul sursă (immediat, direct, indirect și indexat) și respectiv 3 moduri de adresare la operandul destinație (adresarea imediată la destinație reprezintă cod ilegal). Cele 4 moduri de adresare la sursă și respectiv 3 moduri de adresare la destinație generează 12 combinații. Prin urmare, instrucțiunea generică **MOV dest, src** generează 12 instrucțiuni propriu-zise diferite. Din punctul de vedere al codificării, instrucțiunea generică (**MOV**) se codifică în **OPCODE** iar cele 12 combinații diferite se codifică în câmpurile **MAS** și **MAD** din codul mașină al instrucțiunii (figura 4.4).

O instrucțiune generică din clasele B și C generează doar 3 instrucțiuni (combinării) diferite; instrucțiunea generică se codifică în câmpul OPCODE iar cele 3 instrucțiuni propriu-zise (diferite) se codifică în câmpul MAD (figura 4.5, cu AM ilegal la clasa B și respectiv AD ilegal la clasa C).

Instrucțiunile din clasela D nu conțin (în formatul lor) câmpuri „mod de adresare” și prin urmare nu vor produce acest efect de multiplicare (în clasa D, instrucțiunile generice vor fi și instrucțiuni propriu-zise).

Prin urmare, câmpul OPCODE trebuie să asigure:

- codificarea celor 4 clase (A, B, C și respectiv D)
- codificarea instrucțiunilor (generice) în cadrul fiecărei clase.

Pentru codificarea celor 4 clase ar fi necesari 2 biți (cei mai semnificativi 2 biți din OPCODE). Prin urmare, cea mai simplă codificare a celor patru clase ar fi cea redată în tabelul 4.4.

IR₁₅	IR₁₄	Clasa
0	0	A
0	1	B
1	0	C
1	1	D

Tabelul 4.4 Codificarea claselor – versiunea 1

Cu ajutorul celorlalți biți din OPCODE se vor codifica instrucțiunile în cadrul clasei. La clasa A câmpul OPCODE are însă doar 4 biți (IR₁₅÷IR₁₂) iar biții IR₁₅ și IR₁₄ au fost deja alocati pentru codificarea clasei. Rezultă că rămân doar 2 biți (IR₁₃ și IR₁₂) pentru codificarea instrucțiunilor din cadrul acestei clase. Însă, cu doar 2 biți nu pot fi codificate cele 7 instrucțiuni din clasa A. Pentru a codifica 7 instrucțiuni sunt necesari 3 biți. Această problemă nu apare la clasele B, C și D, unde avem câmpuri OPCODE mult mai extinse. Prin urmare, va trebui să revenim la codificarea claselor și să aplicăm un artificiu prin care, pentru clasa A, să utilizăm un singur bit (IR₁₅) pentru codificarea clasei. Ne vor rămâne astfel 3 biți (IR₁₄, IR₁₃ și IR₁₂) pentru a codifica cele 7 instrucțiuni din cadrul acestei clase. Artificiul pe care îl propunem este redat în tabelul 4.5.

IR₁₅	IR₁₄	IR₁₃	CL₁	CL₀	Clasa
0	x	x	0	0	A
1	0	1	0	1	B
1	1	0	1	0	C
1	1	1	1	1	D

Tabelul 4.5 Codificarea claselor – versiunea 2

Din tabelul 4.5 rezultă regulile de selecție a celor 4 clase:

- $IR_{15}=0$ –definește clasa A
- $IR_{15}=1$ –definește (global) clasele B, C și D urmând ca biții IR_{14} și IR_{13} să codifice efectiv (să diferențieze între ele) aceste clase. Din fericire, la clasele B, C și D dispunem de câmpuri OPCODE extinse și ne putem permite să alocăm 3 biți din OPCODE (IR_{15} , IR_{14} și IR_{13}) pentru codificarea acestor trei clase.

Variabilele CL_1 și CL_0 care conform tabelului 4.5 codifică cele 4 clase, pot fi generate cu ajutorul unui *hardware* foarte simplu (figura 4.13).

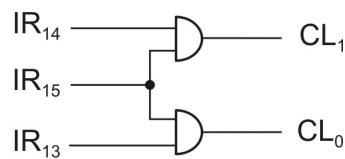


Fig 4.13 Generarea semnalelor care codifică cele 4 clase de instrucțiuni

Pe baza celor stabilite se poate trece la codificarea întregului set de instrucțiuni. Modul de codificare a celor 4 clase și respectiv a instrucțiunilor din cadrul fiecarei clase este redat în figura 4.14.

În figura 4.14, pentru fiecare clasa de instrucțiuni este evidențiată zona de OPCODE utilizată pentru codificarea instrucțiunilor (în cadrul clasei). Astfel:

- $IR_{15}=0$ definește clasa A iar următorii 3 biți din OPCODE (IR_{14} , IR_{13} și IR_{12}) codifică cele 7 instrucțiuni din cadrul clasei A.
- $IR_{15}, IR_{14}, IR_{13}=1,0,1$ definește clasa B iar biții IR_{11} , IR_{10} , IR_9 și IR_8 codifică cele 13 instrucțiuni din cadrul clasei B.
- $IR_{15}, IR_{14}, IR_{13}=1,1,0$ definește clasa C iar biții IR_{11} , IR_{10} , IR_9 și IR_8 codifică cele 10 instrucțiuni din cadrul clasei C.
- $IR_{15}, IR_{14}, IR_{13}=1,1,1$ definește clasa D iar biții IR_{11} , IR_{10} , IR_9 și IR_8 codifică cele 20 instrucțiuni din cadrul clasei D. Observația care urmează clarifică modul de codificare, pe 4 biți, a celor 20 instrucțiuni (!) din clasa D:

CLASA	INSTRUCȚIUNE	IR ₁₅	IR ₁₄	IR ₁₃	IR ₁₂	IR ₁₁	IR ₁₀	IR ₉	IR ₈	IR ₇	IR ₆	IR ₅	IR ₄	IR ₃	IR ₂	IR ₁	IR ₀
A	MOV	0	0	0	0	MAS	RS	MAD	RD								
	ADD	0	0	0	1												
	SUB	0	0	1	0												
	CMP	0	0	1	1												
	AND	0	1	0	0												
	OR	0	1	0	1												
	XOR	0	1	1	0												
B	CLR	1	0	1	0	0	0	0	0	0	0	0	MAD	RD			
	NEG	1	0	1	0	0	0	0	1	0	0	0					
	INC	1	0	1	0	0	0	1	0	0	0	0					
	DEC	1	0	1	0	0	0	1	1	0	0	0					
	ASL	1	0	1	0	0	1	0	0	0	0	0					
	ASR	1	0	1	0	0	1	0	1	0	0	0					
	LSR	1	0	1	0	0	1	1	0	0	0	0					
	ROL	1	0	1	0	0	1	1	1	0	0	0					
	ROR	1	0	1	0	1	0	0	0	0	0	0					
	RLC	1	0	1	0	1	0	0	1	0	0	0					
	RRC	1	0	1	0	1	0	1	0	0	0	0					
	PUSH Ri	1	0	1	0	1	0	1	1	0	0	0					
C	POP Ri	1	0	1	0	1	1	0	0	0	0	0	MAD	RD			
	BEQ	1	1	0	0	0	0	0	0	0	0	0					
	BNE	1	1	0	0	0	0	0	1	0	0	0					
	BMI	1	1	0	0	0	0	1	0	0	0	0					
	BPL	1	1	0	0	0	0	1	1	0	0	0					
	BCS	1	1	0	0	0	1	0	0	0	0	0					
	BCC	1	1	0	0	0	1	0	1	0	0	0					
	BVS	1	1	0	0	0	1	1	0	0	0	0					
	BVC	1	1	0	0	0	1	1	1	0	0	0					
D	JMP	1	1	0	0	1	0	0	0	0	0	0	MAD	RD			
	CALL	1	1	0	0	1	0	0	1	0	0	0					
	CLC	1	1	1	0	0	0	0	0	1	1	1					
	CLZ	1	1	1	0	0	0	0	0	1	1	1					
	CLS	1	1	1	0	0	0	0	0	1	1	1					
	CLV	1	1	1	0	0	0	0	0	1	1	1					
	CCC	1	1	1	0	0	0	0	0	1	1	1					
	SEC	1	1	1	0	0	0	0	1	0	0	0					
	SEZ	1	1	1	0	0	0	0	1	0	0	0					
	SES	1	1	1	0	0	0	0	1	0	0	0					
	SEV	1	1	1	0	0	0	0	1	0	0	0					
	SCC	1	1	1	0	0	0	0	1	0	0	0					
	NOP	1	1	1	0	0	0	1	0	0	0	0					
	HALT	1	1	1	0	0	0	1	1	0	0	0					
	EI	1	1	1	0	0	1	0	0	0	0	0					
	DI	1	1	1	0	0	1	0	1	0	0	0					
	PUSH PC	1	1	1	0	0	1	1	0	0	0	0					
	POP PC	1	1	1	1	0	1	1	0	0	0	0					
	PUSH FLAG	1	1	1	1	1	0	0	0	0	0	0					
	POP FLAG	1	1	1	1	1	0	0	1	0	0	0					
	RET	1	1	1	1	1	0	1	0	0	0	0					
	IRET	1	1	1	1	1	0	1	1	0	0	0					

Fig 4.14 Codificarea instrucțiunilor

Observație:

Să remarcăm că primele 5 instrucțiuni din clasa D (instrucțiunile de resetare a flag-urilor de condiții) sunt codificate cu un **OPCODE** comun pe zona $IR_{15:IR_8}$ (pe octetul cel mai semnificativ din codul instrucțiunii). Diferențierea acestor instrucțiuni este realizată de octetul inferior ($IR_7:IR_0$) unde am codificat de fapt o mască. În microrutina de execuție aferentă acestor instrucțiuni vom realiza operația (4.2).

$$FLAG \leftarrow FLAG \text{ AND } IR_7:IR_0 \quad (4.2)$$

Operandul $IR_7:IR_0$ reprezintă deci masca, care va fi emisă pe DBUS și aplicată pe intrarea ALU cu extensie de semn pe octetul superior (figura 4.15). Valoarea măștii a fost stabilită ținând cont de poziția celor 4 flag-uri (4 bistabili) de condiții în cadrul registrului FLAG (figura 4.16).

Prin extensia de semn valoarea măștii nu se modifică. Prin urmare, se va face o operație **AND** între conținutul registrului FLAG (operandul 1) și masca emisă cu extensie de semn (operandul 2) și rezultatul va fi depus în registrul FLAG. Biții care au valoarea 0 în mască vor determina resetarea biților corespondenți din registrul FLAG. Biții care au valoarea 1 în mască vor lăsa nemodificați biții corespondenți din registrul FLAG. Prin urmare, microrutina de execuție aferentă acestor 5 instrucțiuni va fi una comună (vezieticheta CLC: din microprogramul de emulare - tabelul 4.9), în care se va executa operația (4.2); flag-ul ce va fi resetat va fi decis exclusiv de mască.

O strategie similară vom aplica și pentru următoarele 5 instrucțiuni din clasa D (instrucțiunile de setare a flag-urilor de condiții). În mod analog, aceste 5 instrucțiuni vor avea o microrutină de execuție comună în care se va efectua operația (4.3).

$$FLAG \leftarrow FLAG \text{ OR } IR_7:IR_0 \quad (4.3)$$

Se va face deci o operație **OR** între conținutul registrului FLAG (operandul 1) și masca codificată în octetul inferior al instrucțiunii (care va fi emisă cu extensie de semn pe octetul superior) și rezultatul va fi depus în registrul FLAG. Biții care au valoarea 1 în mască vor determina setarea biților corespondenți din registrul FLAG. Biții care au valoarea 0 în mască vor lăsa nemodificați biții corespondenți din registrul FLAG. Prin urmare, microrutina de execuție aferentă acestor 5 instrucțiuni va putea fi una comună (vezieticheta SEC: din microprogramul de emulare - tabelul 4.9) în care se va efectua operația (4.3); flag-ul ce va fi setat va fi decis exclusiv de mască.

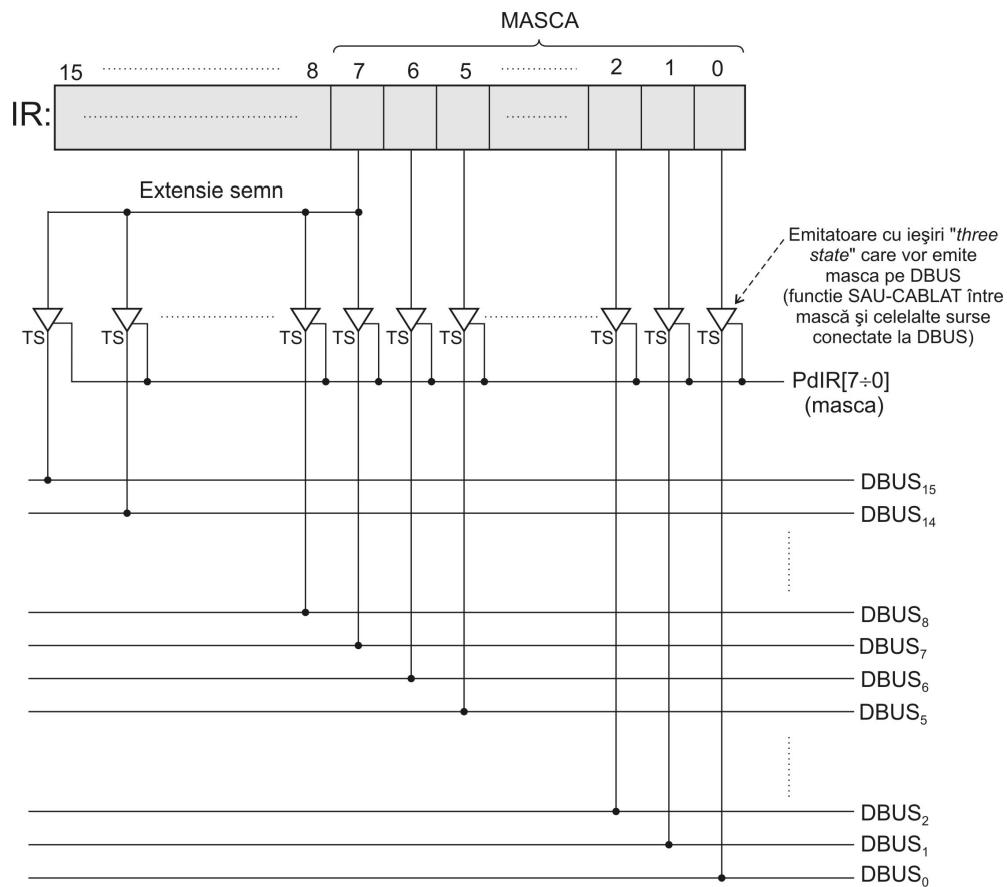


Fig 4.15 Emisia măștii pe DBUS cu extensie de semn pe octetul superior

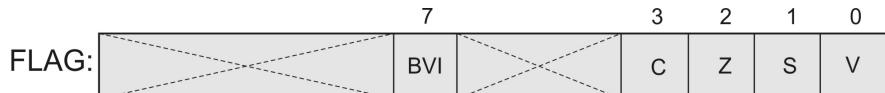


Fig 4.16 Registrul FLAG

Să mai observăm că, pentru clasele B, C și D am utilizat aceeași zonă de OPCODE (zona IR₁₁÷IR₈) pentru codificarea instrucțiunilor (în cadrul clasei). În cadrul microprogramului de emulare, microroutinele de execuție vor fi selectate printr-un salt indexat iar indexul utilizat va fi tocmai câmpul IR₁₁÷IR₈. Rezultă că, pentru cele 3 clase (B, C și D), se va utiliza același index pentru selecția microrutinelor de execuție. Reducerea numărului de indecsii conduce la simplificarea *hardware*-ului care implementează blooul de selecție index.

Observație:

În partea dreaptă a indexului, în general se concatenează un număr de zerouri (vezi 3.4.1.B). Numărul de zerouri concatenate depinde de lungimea microrutinelor pe care le săintește saltul indexat respectiv. Dacă microrutinile au lungimea de 2 microinstructiuni atunci, vom concatena un singur bit de zero în dreapta indexului și salturile se vor executa din 2 în 2 microadrese (microinstructiuni). Dacă microrutinile au lungimea de 4 microinstructiuni atunci, vom concatena 2 biți de zero în dreapta indexului și salturile se vor executa din 4 în 4 microadrese (vezi 3.4.1.B). Dacă microrutinile au lungimea de o singură microinstructiune, atunci nu vom concatena zerouri și prin urmare salturile se vor face pe microadrese (microinstructiuni) succesive.

Pe baza evaluării lungimii microrutinelor de execuție am ajuns la următoarele concluzii:

- Pentru clasa B, cele mai multe microrutine de execuție au lungimea de o singură microinstructiune; pentru selecția acestora vom utiliza $INDEX_5 = IR_{11} \div IR_8$ (fără zerouri concatenate la dreapta; vezi figura 4.17).
- Pentru clasele C și D, cele mai multe microrutine de execuție au lungimea de două microinstructiuni; pentru selecția acestora vom utiliza $INDEX_6 = IR_{11} \div IR_8$, 0 (cu un bit de zero concatenat la dreapta; vezi figura 4.17).

Pentru clasa A, conform figurii 4.14, codificarea instrucțiunilor (în cadrul clasei) este realizată cu alți 3 biți din OPCODE (biții $IR_{14} \div IR_{12}$). Pe de altă parte, cele mai multe microrutine de execuție din clasa A au lungimea de o singură microinstructiune. Prin urmare, pentru selecția acestora vom utiliza $INDEX_4 = IR_{14} \div IR_{12}$ (fără zerouri concatenate la dreapta; vezi figura 4.17).

Se poate concluziona că, valoarea exactă a indecșilor va putea fi stabilită doar după proiectarea microrutinelor din cadrul microprogramului de emulare și după evaluarea lungimii acestora. Doar după această evaluare de lungime se poate decide numărul de zerouri concatenate la dreapta fiecărui index.

În final să menținăm că, pentru codificarea instrucțiunilor există multiple soluții iar proiectantul trebuie să urmeze strategii de optimizare și la acest nivel. Soluția pe care noi am adoptat-o (sintetizată în figura 4.14) vizează simplificarea decodificatorului instrucțiunii și reducerea numărului de indecsi necesari pentru selecția microrutinelor de execuție. Reducerea numărului de indecsi conduce la simplificarea blocului de selecție index.

4.2 Unitatea de control micropogramată

În cele ce urmează vom parcurge etapele de proiectare a unității de control micropogramate. Soluțiile pe care le vom adopta pe parcursul diverselor faze de proiectare vor viza maximizarea performanțelor procesorului CISC definit la 4.1.

4.2.1 Organizarea microprogramului de emulare

Organizarea microprogramului de emulare aferent setului de instrucțiuni definit la 4.1.2 este redată în figura 4.17, unde microrutinile sunt reprezentate simbolic prin segmente de dreaptă.

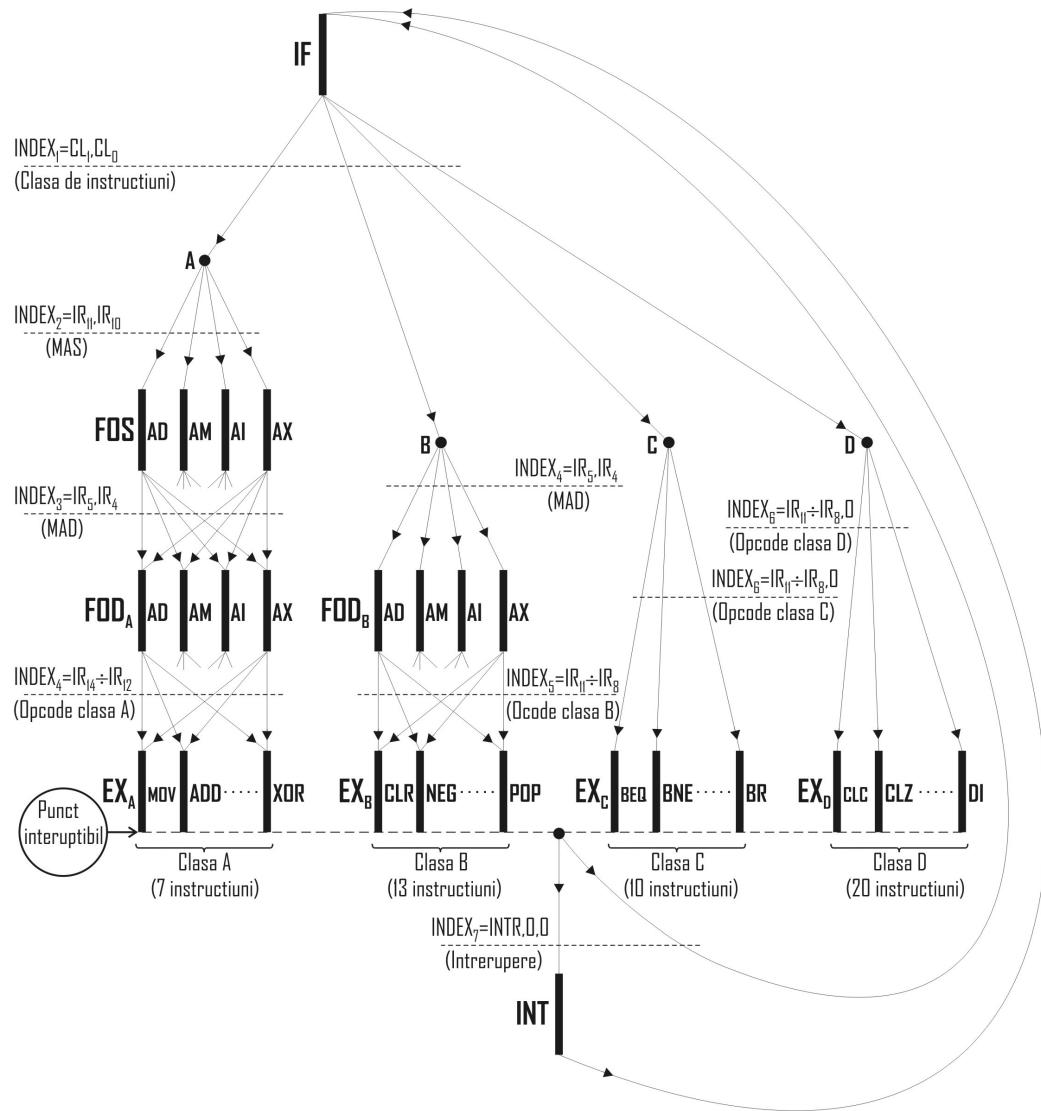


Fig 4.17 Organizarea microprogramului de emulare

Procesarea unei instrucțiuni (emularea instrucțiunii în microcod) începe cu microrutina **Instruction Fetch (IF)**, prin care instrucțiunea este citită din memorie și este încărcată în registrul IR. Selecția locației de memorie se face cu adresa existentă în registrul PC. După încărcarea instrucțiunii în IR urmează decodificarea acesteia. Decodificare înseamnă printre altele și generarea semnalelor CL_1 și CL_0 (figura 4.13) care vor preciza clasa din care face

parte instrucțiunea (vezi tabelul 4.5). În paralel cu decodificarea se realizează și incrementarea cu 2 a registrului PC (microcomanda +2PC). Prin această incrementare se pregătește în PC adresa următoarei instrucțiuni (se pregătește anticipat, adică înainte de execuția instrucțiunii curente).

La ieșirea din microrutina IF apare prima ramificație pe direcțiile A, B, C sau D, în funcție de clasa din care face parte instrucțiunea din IR. Ramificația se va implementa printr-un salt indexat cu indexul $\text{INDEX}_1=\text{CL}_1, \text{CL}_0$.

În continuare, evoluția (traseul în cadrul microprogramului de emulare) depinde de clasa din care face parte instrucțiunea:

- Dacă instrucțiunea este din clasa A (instrucțiune cu 2 operanzi), urmează o microrutină **Fetch Operand Sursă (FOS)** urmată de o microrutină **Fetch Operand Destinație (FOD_A)** și de o microrutină de **Execuție (EX_A)**. La ieșirea din microrutina de execuție se implementează punctul interruptibil (se testează cererea globală de intrerupere INTR). Pentru creșterea vitezei de procesare, punctul interruptibil va fi implementat tot printr-un salt indexat, cu indexul $\text{INDEX}_7=\text{INTR}, 0, 0$.

Există 4 microrutini **FOS** și respectiv 4 microrutini **FOD_A** (cu precizarea că **FOD_A** cu adresare imediată reprezintă de fapt cod ilegal). Cu **FOD_A** am notat colecția de 4 microrutini **Fetch Operand Destinație** utilizate pentru instrucțiunile din clasa A.

Selecția microrutinei **FOS** (una din patru) trebuie făcută în funcție de modul de adresare sursă (MAS) codificat în instrucțiune; selecția se va realiza printr-un salt indexat cu $\text{INDEX}_2=\text{IR}_{11}, \text{IR}_{10}$.

Analog, selecția microrutinei **FOD_A** trebuie făcută în funcție de modul de adresare destinație (MAD) codificat în instrucțiune; selecția se va realiza printr-un salt indexat cu $\text{INDEX}_3=\text{IR}_5, \text{IR}_4$.

Selecția microrutinei **EX_A** (una din 7) trebuie realizată în funcție de OPCODE-ul propriu-zis al instrucțiunii din IR. Prin urmare, selecția se va realiza printr-un salt indexat cu $\text{INDEX}_4=\text{IR}_{14} \div \text{IR}_{12}$.

Să notăm că, microrutina **FOS** selectată, va localiza operandul sursă (conform cu modul de adresare codificat în câmpul MAS), îl va citi și-l va încărca în procesor (în registrul tampon, T). Analog, microrutina **FOD_A** selectată, va localiza operandul destinație (conform cu modul de adresare codificat în câmpul MAD), îl va citi și-l va încărca în procesor (în registrul tampon, MDR). Microrutina **EX_A** selectată, va efectua operația (specificată prin OPCODE) asupra celor 2 operanzi și va depune rezultatul peste operandul destinație.

- Dacă instrucțiunea este din clasa B (instrucțiuni cu un singur operand), urmează o microrutină **Fetch Operand Destinație (FOD_B)** urmată de o microrutină de **Execuție (EX_B)**. La ieșirea din microrutina de execuție se implementează punctul interruptibil (identic cu cel implementat la clasa A: salt indexat, cu indexul $\text{INDEX}_7=\text{INTR}, 0, 0$). Selecția microrutinei **FOD_B** (una din 4) se va realiza tot printr-un salt indexat, cu indexul $\text{INDEX}_3=\text{IR}_5, \text{IR}_4$ (exact ca la

clasa A). Selecția microrutinei **EX_B** (una din 13) se va realiza tot printr-un salt indexat, cu indexul $\text{INDEX}_5=\text{IR}_{11}\div\text{IR}_8$ (OPCODE-ul propriu-zis pentru clasa B).

Cele 4 microrutine **FOD_B** reprezintă copii identice ale microrutinelor **FOD_A**. Am decis implementarea a două seturi identice de microrutine **FOD** (**FOD_A** și **FOD_B**) pentru a simplifica intrarea în microrutinile **EX_A** și respectiv **EX_B**. Dacă am fi utilizat același set de microrutine **FOD** pentru ambele clase de instrucțiuni atunci, la ieșirea din microrutinile **FOD**, ar fi trebuit implementate două mecanisme de ramificare succesive: primul în două direcții (clasa A și respectiv B) și al doilea pe microrutina aferentă instrucțiunii din cadrul clasei. O astfel de ramificare prin două salturi indexate succesive reprezintă evident o soluție mai neperformantă, dacă o comparăm cu soluția adoptată. Dezavantajul soluției adoptate (soluția cu două seturi de microrutine **FOD**) constă într-un consum suplimentar de memorie MPM care, după cum se va vedea la micropogramul de emulare, va fi nesemnificativ.

- Dacă instrucțiunea este din clasa C (instrucțiuni de salt), urmează direct microrutina de **Execuție (EX_C)**. La ieșirea din microrutina de execuție se implementează punctul interruptibil (identic cu cel implementat la clasele A și B: salt indexat, cu indexul $\text{INDEX}_7=\text{INTR}, 0, 0$).

Selecția microrutinei **EX_C** (una din 10) se va realiza tot printr-un salt indexat, cu indexul $\text{INDEX}_5=\text{IR}_{11}\div\text{IR}_8, 0$ (OPCODE-ul propriu-zis pentru clasa C, urmat de un bit de zero deoarece am rezervat o lungime medie de 2 microinstrucțiuni pentru microrutinile **EX_C**).

- Dacă instrucțiunea este din clasa D (instrucțiuni fără operanți), urmează direct microrutina de **Execuție (EX_D)**. La ieșirea din microrutina de execuție se implementează punctul interruptibil (identic cu cel implementat la clasele A, B și C: salt indexat, cu indexul $\text{INDEX}_7=\text{INTR}, 0, 0$).

Selecția microrutinei **EX_D** (una din 20) se va realiza tot printr-un salt indexat, cu indexul $\text{INDEX}_5=\text{IR}_{11}\div\text{IR}_8, 0$ (OPCODE-ul propriu-zis pentru clasa C, urmat de un bit de zero deoarece am rezervat o lungime medie de 2 microinstrucțiuni pentru microrutinile **EX_D**).

Să observăm că instrucțiunile din clasele C și D sunt emulate printr-o succesiune de doar 2 microrutine (**Fetch Instrucțiune** și **Execuție**) și prin urmare, utilizează căi similare pentru emulare în microcod.

Dacă în punctul interruptibil se detectează $\text{INTR}=0$ atunci, după microrutina de execuție, urmează microrutina **Fetch Instrucțiune** (se va face *fetch*-ul următoarei instrucțiuni din cadrul programului). Dacă în punctul interruptibil se detectează $\text{INTR}=1$ atunci microrutina de execuție este urmată de microrutina de întrerupere (INT) care are sarcina de a salva în stivă registrele FLAG și PC și de a lansa în execuție *handler*-ul de întrerupere (*handler*-ul aferent perifericului care a activat întreruperea). Identificarea perifericului (*handler*-ului) se face cu ajutorul vectorului de întrerupere.

4.2.2 Formatul microinstrucțiunii

Pentru îmbunătățirea performanțelor procesorului vom proiecta o microinstrucțiune generală în care vom codifica:

- operații de transfer sincron (transferuri de date interne procesorului)
- operații de transfer asincron (transferuri procesor-memorie)
- succesori simpli sau dublii (cei dublii sunt condiționați și permit ramificații complexe în microcod)

Pentru proiectarea microinstrucțiunii generale vom urma pașii definiți la 3.12 (figura 3.47 – varianta 1 de lucru). Utilizarea variantei 1 de lucru este justificată de faptul că procesorul CISC (figura 4.1) dispune de registre *buffer* pentru interfațarea cu memoria:

- registrul ADR – *buffer* de adrese
- registrul MDR – *buffer* de date

Prin introducerea registrului MDR, operațiile de scriere în memorie se simplifică în sensul că datele ce vor fi scrise în memorie vor fi preluate întotdeauna din MDR. Drept consecință, câmpurile **SURSA SBUS**, **SURSA DBUS**, **OPERAȚIE ALU** și **SURSA RBUS** (cele care provin din microinstrucțiunea de transfer asincron) vor fi eliminate (pasul 3, figura 3.47 – varianta 1). De fapt nu va fi preluat nici câmpul **SURSA RBUS** din microinstrucțiunea de transfer sincron deoarece procesorul CISC (figura 4.1) are o singură sursă pe RBUS (unitatea ALU). În aceste condiții, ieșirile ALU pot fi conectate galvanic la liniile RBUS-ului deoarece, existând o singură sursă, pe RBUS nu pot să apară conflicte. Drept consecință, câmpul **SURSA RBUS** devine inutil.

După cum precizam la 3.12, la finele pasului 3 se obține versiunea 1 de microinstrucțiune generală dedicată procesorului didactic. Pentru a obține microinstrucțiunea generală dedicată procesorului CISC definit la 4.1 mai trebuie parcurs un ultim pas (pasul 4) pe care îl vom denumi **transpunere**; microinstrucțiunea generală trebuie transpusă de pe o arhitectură (procesorul didactic) pe o altă arhitectură (procesorul CISC). În figura 4.18 este redat rezultatul acestei transpuneri.

Referitor la structura microinstrucții generale din figura 4.18, câmpurile **SURSA SBUS**, **SURSA DBUS**, **OPERAȚIE ALU** și **DESTINAȚIE RBUS** provin din microinstrucțiunea de transfer sincron. Cu ajutorul comenziilor codificate în aceste câmpuri se controlează transferurile de date registru-registru (transferurile interne). Informația transferată traversează întotdeauna unitatea ALU și la nevoie, poate fi procesată în ALU.

Procesorul CISC pe care-l proiectăm are o structură orientată pe registre generale. Să observăm că microcomenzile de deplasare și rotire le-am codificat în câmpul **OPERAȚIE ALU**. Prin urmare, operațiile de deplasare și rotire le vom implementa în ALU (vezi cele 2 observații de la 2.7.2.D).

Fig. 4.18 Microinstrucțiunea generală aferentă procesorului CISC

Câmpul **OPERAȚII CU MEMORIA** provine din microinstrucțiunea de transfer asincron. Prin urmare, în paralel cu un transfer sincron (registru-registru) se poate activa și o operație de citire din sau de scriere în memorie. Dacă se activează o scriere (prin microcomanda WRITE), atunci va fi preluată data din MDR și va fi scrisă în locația de memorie selectată de adresa existentă în ADR. Dacă se activează o citire prin microcomanda READ, atunci se vor citi informațiile stocate în locația selectată de adresa prezentă în ADR și informațiile citite vor fi încărcate în registrul MDR; astfel se realizează *fetch*-ul unor date din memorie (operand imediat sau index). Dacă se activează o citire prin microcomanda IFCH, atunci se vor citi informațiile stocate în locația selectată de adresa prezentă în ADR și informațiile citite vor fi încărcate în registrul IR; astfel se realizează *fetch*-ul unei instrucțiuni din memorie.

Câmpul **ALTE OPERAȚII** a rezultat prin unificarea câmpurilor OTHER OPERATIONS (din microinstrucțiunea de transfer sincron) și respectiv SHIFT & OTHER OPERATIONS (din microinstrucțiunea de transfer asincron), urmată de transpunerea de pe o arhitectură (procesorul didactic) pe altă arhitectură (procesorul CISC). Evident că setul de comenzi „*other*” ce vor fi codificate în acest câmp nu va putea fi definitivat decât după proiectarea efectivă a microprogramului de emulare. Din microprogram se va putea extinge lista comenziilor din categoria „*other*” și toate comenziile din listă vor trebui codificate în câmpul **ALTE OPERAȚII** al microinstrucțiunii generale. Este momentul în care se poate deduce exact și numărul de biți din compoziția acestui câmp.

În sfârșit, câmpurile **SUCESOR**, **SELECTIE INDEX**, \bar{T}/F și **MICROADRESA DE SALT** provin din microinstrucțiunea de ramificație. Numărul de biți din compoziția câmpului **MICROADRESA DE SALT** va putea fi stabilit numai după proiectarea microprogramului de emulare deoarece depinde de lungimea acestuia. Dacă microprogramul va avea lungimea de 2^n microinstrucțiuni (2^n locații în MPM) atunci câmpul **MICROADRESA DE SALT** va avea n biți în compoziția sa. Câmpurile **SUCESOR** și **SELECTIE INDEX** vor putea fi definite tot după proiectarea microcodului (microprogramului de emulare).

Pentru stabilirea succesorilor va trebui să avem lista tuturor condițiilor testate în microcod. Înainte de proiectarea microcodului, putem doar intui condițiile testate dar nu putem fi siguri că lista acestora este completă. Pentru procesorul nostru CISC, este clar că în microroutinele de execuție aferente instrucțiunilor de *branch* (de salt condiționat) vom testa cele 4 *flag*-uri de condiții: C, Z, S și V. Este clar de asemenea că vom testa condițiile de excepție ACLOW (căderea tensiunii de alimentare) și CIL (cod ilegal). Pentru creșterea vitezei de operare a procesorului (viteza de emulare în microcod a instrucțiunilor), punctul interrupțional vom implementa un salt indexat cu $INDEX_7=INTR$, 0, 0 (vezi fig 4.17). În această situație, condiția INTR nu va mai trebui testată și prin urmare o vom elimina din listă.

Pornim deci de la premisa că lista de condiții ce vor trebui testate este: ACLOW, CIL, C, Z, S, și V. Având stabilită lista condițiilor testate și ținând cont că bitul $MIR_7=\bar{T}/F$ specifică modul de test a condiției selectate (test pe fals sau pe adevarat), successorii pot fi definiți așa cum indică tabelul 4.6.

O nouă特ă de successorii definiți la procesorul didactic (tabelul 3.2), ar fi apariția successorului STEP (explicit) în tabelul 4.6. Astă e, la procesorul didactic successorul STEP a fost un successor implicit al microinstrucțiunilor de transfer sincron și asincron; el nu a fost codificat explicit în aceste microinstrucțiuni ci a fost dedus implicit de către SEQ, pornind de la tipul microinstrucțiunii (vezi blocul (6) din fig 3.7). La procesorul CISC

operăm cu o microinstrucțiune generală. În această microinstrucțiune trebuie să codificăm un succesor STEP explicit; fără un STEP explicit nu ar fi posibilă procesarea sevențială a microinstrucțiunilor din interiorul unei microrutine.

După proiectarea microcodului, va fi stabilită cu exactitate lista de condiții testate (versiunea finală). Dacă această listă definitivă prezintă modificări față de lista inițială atunci vor fi revizuiți și succesorii codificați în microinstrucțiune (va fi revizuit tabelul 4.6).

MIR ₁₃₋₁₁	Mnemonica succesorului	Funcția f testată
000	STEP	f=MIR ₇ ; rezultă g=MIR ₇ XOR MIR ₇ =0
001	JUMPI μADR+INDEX	f= MIR ₇ ; rezultă g= MIR ₇ XOR MIR ₇ =1
010	IF (N)ACLOW JUMPI μADR+INDEX else STEP	f=ACLOW; rezultă g=ACLOW XOR MIR ₇
011	IF (N)CIL JUMPI μADR+INDEX else STEP	f=CIL; rezultă g=CIL XOR MIR ₇
100	IF (N)C JUMPI μADR+INDEX else STEP	f=C; rezultă g=C XOR MIR ₇
101	IF (N)Z JUMPI μADR+INDEX else STEP	f=Z; rezultă g=Z XOR MIR ₇
110	IF (N)S JUMPI μADR+INDEX else STEP	f=S; rezultă g=S XOR MIR ₇
111	IF (N)V JUMPI μADR+INDEX else STEP	f=V; rezultă g=V XOR MIR ₇

Tabelul 4.6 Succesorii JUMPI

Ramificațiile în mai mult de 2 direcții se implementează în micocod prin salturi indexate. La succesorul JUMPI, câmpul SELECTIE INDEX are sarcina de a selecta indexul dorit. În figura 4.17 s-a stabilit structura microprogramului de emulare (microcodului) și pe baza aceastei figuri se poate elabora lista inecșilor utilizati. Această listă este sintetizată în tabelul 4.7, unde este redată și valoarea (binară) a fiecărui index. Valoarea a fost stabilită pornind de la indecșii definiți în figura 4.17. Dacă după proiectarea microcodului se va dovedi necesară redefinirea unuia sau mai multor indecși, atunci se va reveni și se va redefini și tabelul 4.7.

Să observăm că primul index din tabelul 4.7 (INDEX₀) are valoarea 0. Cu ajutorul acestui index vom converti succesorii JUMPI din tabelul 4.6, în succesi JUMP echivalenți (vezi explicațiile relative la tabelele 3.1 și 3.2 de la 3.4.1). Succesi JUMP sunt prezenți în tabelul 4.8.

Microinstrucțiunea generală din figura 4.18 va fi o microinstrucțiune performantă, care va permite activarea unui set amplu de operații paralele. În varianta cea mai completă, setul poate conține:

- o operație registru-registru (care poate include și procesare efectivă în ALU).
- o operație de transfer cu memoria (citire sau scriere)
- o operație din categoria „alte operații”
- o ramificație implementată prin orice tip de succesor

MIR _{10÷8}	Denumire INDEX	Valoare INDEX (valoare binară)							Semnificație
		IND ₆	IND ₅	IND ₄	IND ₃	IND ₂	IND ₁	IND ₀	
000	INDEX ₀	0	0	0	0	0	0	0	Conversie JUMPI în JUMP
001	INDEX ₁	0	0	0	0	0	CL ₁	CL ₀	Cele 4 clase de instrucțiuni
010	INDEX ₂	0	0	0	0	0	IR ₁₁	IR ₁₀	Modul de adresare sursa
011	INDEX ₃	0	0	0	0	0	IR ₅	IR ₄	Modul de adresare destinație
100	INDEX ₄	0	0	0	0	IR ₁₄	IR ₁₃	IR ₁₂	OPCODE clasa A
101	INDEX ₅	0	0	0	IR ₁₁	IR ₁₀	IR ₉	IR ₈	OPCODE clasa B
110	INDEX ₆	0	0	IR ₁₁	IR ₁₀	IR ₉	IR ₈	0	OPCODE clasele C și D
111	INDEX ₇	0	0	0	0	INTR	0	0	Cererea globală de întrerupere

Tabelul 4.7 Indecșii definiți

MIR _{13÷11}	Mnemonica succesorului	Funcția f testată
000	STEP	f=MIR ₇ ; rezultă g=MIR ₇ XOR MIR ₇ =0
001	JUMP μADR	f=MIR ₇ ; rezultă g=MIR ₇ XOR MIR ₇ =1
010	IF (N)ACLOW JUMP μADR else STEP	f=ACLOW; rezultă g=ACLOW XOR MIR ₇
011	IF (N)CIL JUMP μADR else STEP	f=CIL; rezultă g=CIL XOR MIR ₇
100	IF (N)C JUMP μADR else STEP	f=C; rezultă g=C XOR MIR ₇
101	IF (N)Z JUMP μADR else STEP	f=Z; rezultă g=Z XOR MIR ₇
110	IF (N)S JUMP μADR else STEP	f=S; rezultă g=S XOR MIR ₇
111	IF (N)V JUMP μADR else STEP	f=V; rezultă g=V XOR MIR ₇

Tabelul 4.8 Succesorii JUMP

Să revenim la ultimul pas al algoritmului de elaborare a microinstrucțiunii generale (pasul 4 = **transpunere**). La finele pasului 3 definit la 3.12 (figura 3.47 – varianta 1 de lucru) s-a obținut microinstrucțiunea generală aferentă procesorului didactic. Pasul 4 presupune **transpunerea** microinstrucțiunii generale de pe o arhitectură pe alta (de pe arhitectura procesorului didactic pe arhitectura procesorului CISC). Această transpunere trebuie realizată pentru fiecare câmp al microinstrucțiunii generale obținută la finele pasului 3. În figura 4.19, este exemplificată operația de transpunere pentru câmpul SURSA SBUS.

Procesorul Didactic		Transpunere SURSA SBUS		Procesorul CISC	
SURSA SBUS (MIR ₁₇₊₁₅)	Microcomenzi codificate			SURSA SBUS (MIR ₃₅₊₃₂)	Microcomenzi codificate
000	NONE			0000	NONE
001	PdIR[OP]			0001	PdFLAGS
010	PdIR[IND]			0010	PdRGs
011	PdA			0011	PdSPs
100	PdSP			0100	PdT _s
101	Pd0s			0101	PdT̄s
110	Pd-1s			0110	PdPCs
111	-			0111	PdIVRs
				1000	PdADRs
				1001	PdMDRs
				1010	PdIR[7÷0]s
				1011	Pd0s
				1100	Pd-1s
				1101	-
				1110	-
				1111	-

Fig 4.19 Operația de transpunere a câmpului SURSA SBUS

Observăm din figura 4.19 că la procesorul didactic, câmpul SURSA SBUS are 3 biți (MIR₁₇₊₁₅) și codifică microcomenziile care emit pe SBUS registrele IR[OP], IR[IND], A, SP și constantele 0 și -1 (vezi figurile 3.3 și 2.2). La procesorul CISC avem alte registre conectate la SBUS (vezi figura 4.1). Prin urmare, câmpul SURSA SBUS trebuie să codifice comenzi aferente acestor registre (PdFLAGS, PdRGs, PdSPs, PdT_s, PdT̄s, PdPCs, PdIVRs, PdADRs, PdMDRs, PdIR[7÷0]s) și comenzi de predare a unor constante (Pd0s și Pd-1s). Am utilizat indicele „s” deoarece, la procesorul CISC, comenzi similare vor fi codificate și în câmpul SURSA DBUS; pentru acestea vom utiliza indicele „D”. Pe baza acestor indici vom diferenția comenziile codificate în câmpul SURSA SBUS de cele codificate în câmpul SURSA DBUS.

Pe de altă parte, la procesorul CISC avem mai multe registre conectate la SBUS. Rezultă astfel necesitatea extinderii câmpului SURSA SBUS de la 3 la 4 biți (MIR₃₅₊₃₂).

Constantele nu pot fi stabilite concret decât după proiectarea microprogramului de emulare. Nici comenziile de tipul PdT̄s, care revindică și inversarea (bit cu bit) a conținutului registrului specificat, nu pot fi stabilite concret decât după proiectarea microcodului. Prin urmare, prin transpunere rezultă o versiune aproximativă (nefinalizată) a câmpului SURSA SBUS. Aceasta va putea fi definitivată doar după definitivarea microprogramului de emulare. Doar după definitivarea microcodului pot fi inventariate toate comenziile care trebuie codificate în câmpul SURSA SBUS.

În final trebuie să menționăm că această operație de transpunere trebuie efectuată pentru fiecare câmp din microinstrucțiunea generală obținută la finele pasului 3 de la 3.12 (figura 3.47 – varianta 1). Astfel a rezultat microinstrucțiunea generală din figura 4.18. Cu această microinstrucțiune vom proiecta microprogramul de emulare aferent procesorului CISC.

4.2.3 Implementarea și controlul regiszrelor generale

Procesorul CISC conține 16 registre generale ($R_0 \div R_{15}$) pe 16 biți. După cum se observă din schema bloc a procesorului (fig 4.1), setul de registre (REGISTER FILE) este conectat la cele 3 busuri interne. Schema de interconectare sugerează că blocul REGISTER FILE acceptă 3 tipuri de accese:

- citirea unui regiszru pe SBUS
- citirea unui regiszru pe DBUS
- scrierea (încărcarea) într-un regiszru a datelor preluate de pe RBUS

Pentru cele 3 operații, în microinstructiunea generală din figura 4.18, am prevăzut 3 microcomenzi dedicate:

- $PdRG_S$ –emite conținutul unui regiszru pe SBUS
- $PdRG_D$ –emite conținutul unui regiszru pe DBUS
- $PmRG$ –scrie într-un regiszru datele preluate de pe RBUS

Microcomanda $PdRG_S$ a fost codificată în câmpul SURSA SBUS, microcomanda $PdRG_D$ a fost codificată în câmpul SURSA DBUS iar microcomanda $PmRG$ a fost codificată în câmpul DESTINAȚIE RBUS (fig 4.18). Selecția regiszrelor asupra cărora vor acționa cele 3 microcomenzi trebuie implementată în hardware.

Pentru a permite accese concurente, blocul de REGISTRE GENERALE trebuie implementat într-o structură multiport, similară celei din figura 1.18. Structura multiport trebuie să disponă de două porturi de citire și un port de scriere. În figura 4.20 este prezentată structura multiport care oferă aceste facilități.

Adresa AC_{3A+0A} selectează regiszrul ce va fi citit pe portul de citire **A**, adresa AC_{3B+0B} selectează regiszrul ce va fi citit pe portul de citire **B** iar adresa AS_{3+0} selectează regiszrul ce va fi scris pe portul de scriere; vor fi scrise datele preluate de pe RBUS. Cele trei seturi de intrări de adrese sunt acționate de câmpurile RS și respectiv RD din codul instructiunii curente; instructiunea curentă este instructiunea aflată în execuție (instructiunea al cărei cod este prezent în regiszrul IR).

Urmărind schema din figura 4.20 putem concluziona:

- regiszrul ce va fi citit pe portul **A** (al cărui conținut va fi emis pe SBUS prin activarea microcomenzi $PdRG_S$) va fi întotdeauna regiszrul sursă specificat de instructiunea curentă.
- regiszrul ce va fi citit pe portul **B** (al cărui conținut va fi emis pe DBUS prin activarea microcomenzi $PdRG_D$) va fi întotdeauna regiszrul destinație specificat de instructiunea curentă.
- regiszrul ce va fi scris pe portul de scriere (în care, prin activarea microcomenzi $PmRG$, vor fi înscrise datele preluate de pe RBUS) va fi întotdeauna regiszrul destinație specificat de instructiunea curentă.

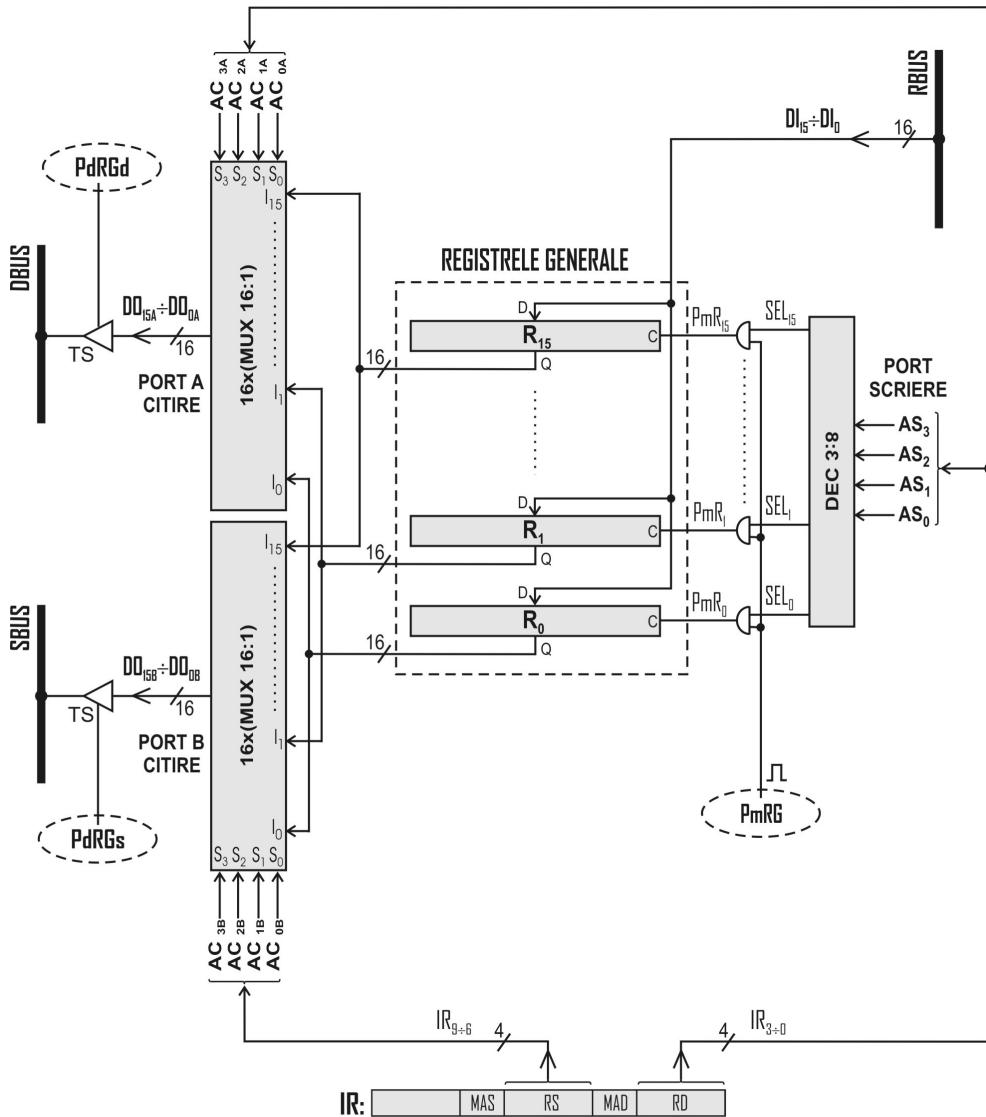


Fig 4.20..Structura multiport aferentă setului de registre generale

Evident că, pe parcursul procesării unei instrucțiuni cu 2 operanzi (pe parcursul emulării instrucțiunii în microcod), registrele generale ce vor trebui accesate vor fi registrul sursă și registrul destinație; registrul sursă va fi utilizat (citit) în microrutina FOS (*Fetch Operand Sursă*), registrul destinație va fi utilizat (citit) în microrutina FOD (*Fetch Operand Destinație*) și reutilizat (scris) în microrutina EX (de Execuție), când în registrul destinație trebuie depus rezultatul. Pe toată durata emulării instrucțiunii, registrele sursă și respectiv

destinație sunt selectate deoarece codul instrucțiunii este prezent în IR pe toată durata ciclului instrucțiunii; acest cod va fi evacuat din IR doar în momentul în care va fi suprascris cu codul următoarei instrucțiuni (când se face *fetch*-ul următoarei instrucțiuni din cadrul programului). Având în vedere că registrele sursă și respectiv destinație sunt selectate pe toată durata emulării instrucțiunii, rezultă că din microcod vom activa, atunci când se va dovedi necesar, doar microcomenzile:

- PdRG_S –pentru emisia pe SBUS a conținutului registrului sursă
- PdRG_D –pentru emisia pe DBUS a conținutului registrului destinație
- PmRG –pentru scrierea în registrul destinație a datelor preluate de pe RBUS

Instrucțiunile cu un singur operand și instrucțiunile de salt utilizează un singur registru și anume registrul destinație. Schema de selecție a regisrelor generale este aceeași doar că, pe parcursul emulării unei asemenea instrucțiuni, va fi utilizat doar registrul destinație.

Instrucțiunile fără operanți nu operează cu registrele generale. Prin urmare, schema de selecție a regisrelor generale din figura 4.20 va fi neutilitată pe parcursul emulării unei asemenea instrucțiuni. De fapt, se vor selecta fals două registre (unul sursă și unul destinație) dar acestea nu vor fi utilizate deoarece, pe parcursul emulării unei asemenea instrucțiuni, nu vor fi activate niciuna dintre cele trei microcomenzi care acionează asupra regisrelor generale (PdRG_S, PdRG_D și respectiv PmRG).

În concluzie, schema din figura 4.20 ne permite să gestionăm prin doar 3 microcomenzi toate operațiile cu regisrele generale.

Observații:

1. Dacă analizăm schema bloc aferentă procesorului CISC (figura 4.1), constatăm că și regisrele speciale (FLAG, SP, T, PC, IVR, ADR, și MDR) vor avea alocate tot căte 3 microcomenzi în microinstrucțiunea generală (similitudine perfectă cu blocul de REGISTRE GENERALE).

Dacă ne referim de exemplu la regiszrul SP, microcomenzi dedicate acestuia sunt:

- PdSP_S –pentru emisia pe SBUS a conținutului registrului SP
- PdSP_D –pentru emisia pe DBUS a conținutului registrului SP
- PmSP –pentru scrierea în regiszrul SP a datelor preluate de pe RBUS

Microcomanda PdSP_S a fost codificată în câmpul SURSA SBUS, microcomanda PdSP_D a fost codificată în câmpul SURSA DBUS iar microcomanda PmSP a fost codificată în câmpul DESTINAȚIE RBUS (fig 4.18). Similar cu regisrele generale, toate regisrele speciale au alocate 3 astfel de microcomenzi în microinstrucțiunea generală.

2. În câmpurile **SURSA SBUS** și **SURSA DBUS** apar suplimentar microcomenzi prin care se generează constante pe **SBUS** și respectiv **DBUS**:
 - **Pd0_S** –emite constanta 0 pe **SBUS**
 - **Pd0_D** –emite constanta 0 pe **DBUS**
 - **Pd-1_S** –emite constanta -1 pe **SBUS**
 - **Pd-1_D** –emite constanta -1 pe **DBUS**

3. Relativ la registrele **T** și **MDR** apar suplimentar comenzi **Pd \bar{T} _S** și respectiv **Pd \bar{MDR} _D** care emit (pe **SBUS** și respectiv **DBUS**) conținutul registrului specificat, inversat bit cu bit (complementul față de 1).

4. Prin intermediul blocului de multiplexoare 2:1 din figura 4.1, registrul **FLAG** poate fi încărcat pe două căi:
 - încărcare paralelă cu datele preluate de pe **RBUS**. Această operație este realizată de microcomanda **PmFLAG**, codificată în câmpul **DESTINAȚIE RBUS**.
 - încărcare în flag-urile de condiții (în bistabilii **C**, **Z**, **S** și **V**) a semnalelor de condiții generate de unitatea **ALU** (**C_{out}**, **Z_R**, **S_R** și **DCR**). Această operație de setare a flag-urilor în acord cu rezultatul trebuie realizată în microroutinele de execuție aferente instrucțiunilor aritmetice și logice. Pentru această operație de setare a flag-urilor am prevăzut două microcomenzi (codificate în câmpul **ALTE OPERAȚII**):
 - **PdCOND_A** –încarcă toate flag-urile (**C**, **Z**, **S** și **V**) cu semnalele de condiții furnizate de unitatea **ALU**; va fi utilizată la instrucțiunile aritmetice.
 - **PdCOND_L** –încarcă doar flag-urile **Z** și **S**; va fi utilizată la instrucțiunile logice. Astă e, la instrucțiunile logice nu se poate obține niciodată **Carry=1** și **Overflow=1**; ca atare flag-urile **C** și **V** nu se poziționează.

5. Pe intrarea registrului **MDR** regăsim de asemenea un bloc de 16 multiplexoare 2:1 (figura 4.1). Prin urmare, și registrul **MDR** poate fi încărcat pe două căi:
 - încărcare paralelă cu datele preluate de pe **RBUS**. Această operație este realizată de microcomanda **PmMDR** codificată în câmpul **DESTINAȚIE RBUS**.
 - Încărcare paralelă cu datele citite din memorie. Această operație este realizată de microcomanda **READ** codificată în câmpul **OPERAȚII CU MEMORIA**.

Cu READ se realizează fetch-ul operanziilor din memorie, în timp ce cu IFCH se realizează fetch-ul instrucțiunilor din memorie. La nivelul memoriei, microcomenziile IFCH și READ efectuează aceeași operație: citirea informațiilor memorate în locația de memorie selectată de adresa existentă în registrul ADR. Diferența între IFCH și READ apare doar la nivelul procesorului (la nivelul destinatarului informațiilor citite din memorie):

- microcomanda IFCH încarcă informația extrasă din memorie în registrul IR (Fetch Instrucțiune).
- microcomanda READ încarcă informația extrasă din memorie (prin intermediul setului de 16 multiplexoare 2:1) în registrul MDR (Fetch Operand).

4.2.4 Proiectarea microsevențiatorului

Microsevențiatorul (automatul SEQ) generează comenzi interne BGC-ului micropogramat, comenzi prin care controlează fetch-ul și execuția microinstrucțiunilor.

A. Organograma de funcționare a microsevențiatorului

La 3.5.1 a fost prezentată organograma de funcționare aferentă automatului SEQ dedicat procesorului didactic. Remintim faptul că la procesorul didactic am definit 3 tipuri de microinstrucțiuni iar automatul SEQ a fost proiectat astfel încât să controleze procesarea microinstrucțiunilor, ținând cont de tipul acestora (vezi blocurile (2) și (7) din figura 3.7). Pentru procesorul nostru CISC am definit un singur tip de microinstrucțiune. Această microinstrucțiune generală, semnificativ mai performantă, cumulează toate capabilitățile celor 3 tipuri de microinstrucțiuni definite le procesorul didactic.

În aceste condiții, microsevențiatorul aferent procesorului CISC va avea o misiune ușor simplificată deoarece nu va mai trebui să decodifice tipul microinstrucțiunii și să inițieze acțiuni dependente de tipul microinstrucțiunii. Prin urmare, blocurile (2) și (7) din figura 3.7 nu vor mai exista în organograma microsevențiatorului aferent procesorului CISC. Vom opera de asemenea unele simplificări și în partea de organigramă care se referă la starea 3 a automatului SEQ. Aceste simplificări vizează reducerea numărului de ieșiri din automat și vor fi posibile dacă introducem un hardware adițional foarte simplu responsabil cu generarea acestor ieșiri (comenzi). În figura 4.21 redăm organograma microsevențiatorului aferent procesorului CISC.

Această organigramă trebuie explicitată prin raportare la:

- schema de principiu a unității de control micropogramate (fig 3.1).
- organograma microsevențiatorului aferent procesorului didactic (fig 3.7)

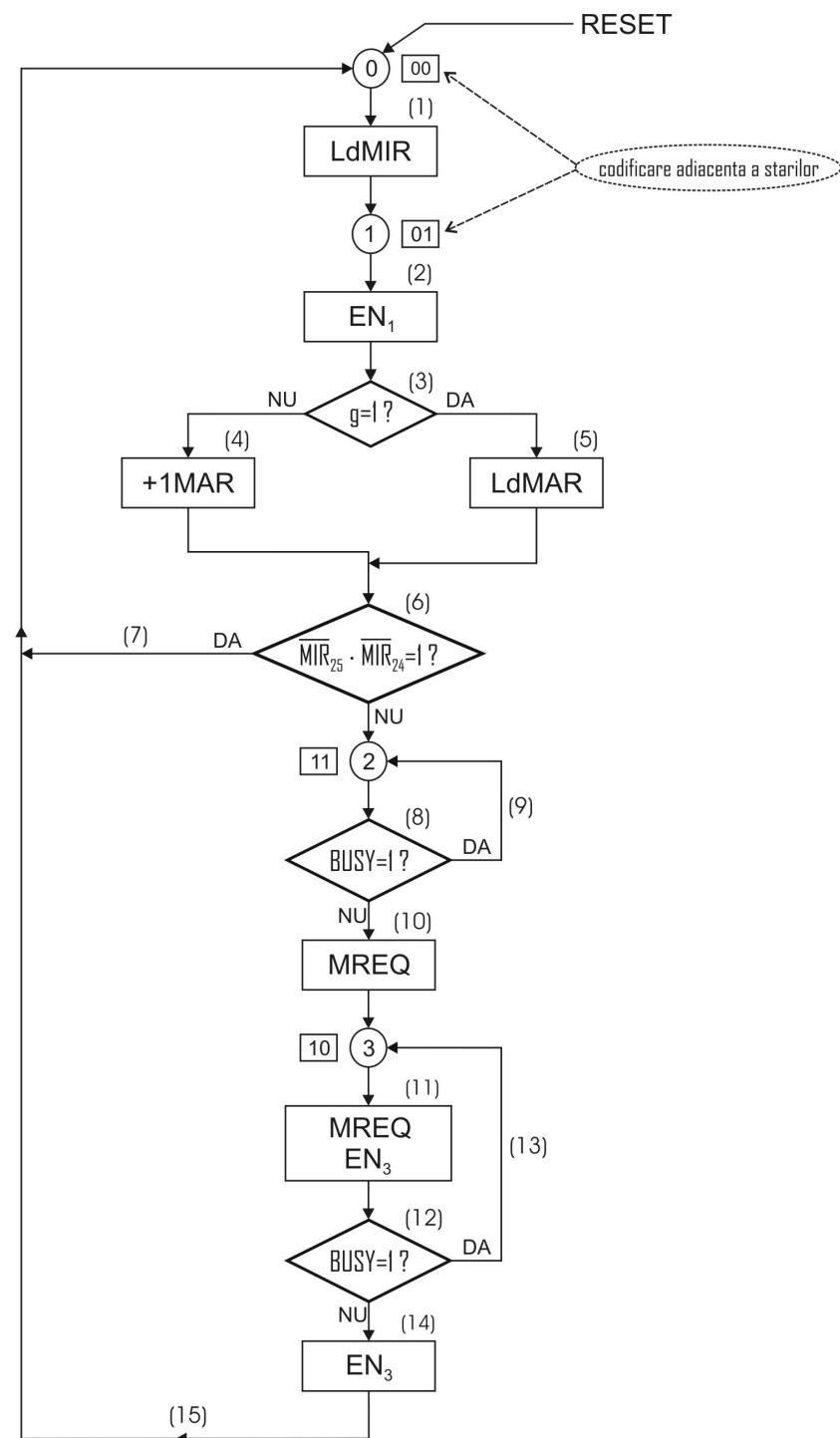


Fig 4.21..Organograma automatului SEQ aferent procesorului CISC

ORGANIZAREA ȘI PROIECTAREA CALCULATOARELOR

Din organigrama prezentată în figura 4.21 constatăm că automatul SEQ aferent procesorului CISC parcurge:

- 2 stări (stările 0 și 1) în cazul în care controlează o microinstrucțiune care are codificat NOP în câmpul OPERAȚII CU MEMORIA (care nu activează o comandă efectivă de citire din sau de scriere în memorie).
- 4 stări (stările 0, 1, 2 și 3) în cazul în care controlează o microinstrucțiune care inițiază efectiv o operație de citire din sau de scriere în memorie (printr-o comandă IFCH, READ sau WRITE codificată în câmpul OPERAȚII CU MEMORIA). În acest caz, în stările 2 și 3 SEQ controlează transferul procesor-memorie (operația de citire din sau de scriere în memorie).

În starea 0, automatul SEQ activează comanda LdMIR (blocul (1) din figura 4.21) prin care încarcă în registrul MIR microinstrucțiunea selectată în MPM de adresa existentă în registrul MAR (*fetch* -ul microinstrucțiunii). Microinstrucțiunea extrasă din MPM și încărcată în registrul MIR devine astfel microinstrucțiune curentă.

În starea 1, SEQ activează comanda EN₁ (blocul (2) din figura 4.21) prin care validează decodificatoarele de microcomenzi (EN₁=enable din starea 1). Prin validarea decodificatoarelor se vor activa toate microcomenzile codificate în microinstrucțiunea curentă iar activarea acestor microcomenzi conduce la efectuarea operațiilor elementare controlate de către microinstrucțiunea curentă.

Tot în starea 1, prin testarea funcției globale de ramificație (blocul (3) din figura 4.21), SEQ decide succesoarea microinstrucțiunii curente:

- dacă în microinstrucțiunea curentă este codificat succesorul **STEP** atunci SEQ va sesiza g=0 (vezi tabelele 4.6 și 4.8) și va activa comanda +1MAR (blocul (4) din figura 4.21). În cazul succesorului STEP se execută deci operația:

$$\text{MAR} \leftarrow \text{MAR} + 1 \quad (4.4)$$

Prin incrementarea registrului MAR, se va trece secvențial la următoarea microinstrucțiune.

Tot comanda +1MAR va fi activată, dacă în microinstrucțiunea curentă este codificat un succesor condiționat de forma **IF CONDIȚIE JUMPI μADR else STEP** (tabelul 4.6) sau **IF CONDIȚIE JUMP μADR else STEP** (tabelul 4.8) și dacă condiția testată este neîndeplinită. Dacă condiția testată este neîndeplinită, funcția globală de ramificație (g) va avea valoarea 0 și prin urmare, se va executa succesorul STEP.

- dacă în microinstrucțiunea curentă este codificat succesorul **JUMPI μADR** (tabelul 4.6) sau **JUMP μADR** (tabelul 4.8) atunci SEQ va sesiza g=1 și va activa comanda LdMAR (blocul (5) din figura 4.21). Se realizează deci o încărcare paralelă în registrul MAR. În cazul succesorului JUMPI se execută operația:

$$\text{MAR} \leftarrow \text{MIR}_{6+0} + \text{INDEX} \mid \text{selectat de MIR10+8} \quad (4.5)$$

iar în cazul succesorului JUMP se execută operația:

$$\text{MAR} \leftarrow \text{MIR}_{6 \div 0} \quad (4.6)$$

deoarece indexul selectat va avea valoarea zero (vezi $\text{INDEX}_0=0$ în tabelul 4.7). Ambele operații presupun încărcare paralelă în registrul MAR.

Tot comanda LdMAR va fi activată dacă în microinstrucțiunea curentă este codificat un succesor condiționat de forma **IF CONDIȚIE JUMPI μADR else STEP** (tabelul 4.6) sau **IF CONDIȚIE JUMP μADR else STEP** (tabelul 4.8) și dacă condiția testată este îndeplinită. Dacă condiția testată este îndeplinită, atunci funcția globală de ramificație (g) va avea valoarea 1 și prin urmare, se va executa succesorul JUMPI (respectiv JUMP). După cum am mai menționat JUMP rezultă automat din JUMPI, dacă indexul selectat este $\text{INDEX}_0=0$.

Tot în starea 1, se testează condiția $\overline{\text{MIR}}_{19} \cdot \overline{\text{MIR}}_{18}$:

- dacă $\overline{\text{MIR}}_{19} \cdot \overline{\text{MIR}}_{18} = 1$ înseamnă că microinstrucțiunea curentă nu lansează nici o operație cu memoria (are NOP codificat în câmpul OPERAȚII CU MEMORIA) În acest caz, din starea 1, automatul SEQ revine în starea 0 (tranziția (7) din organograma 4.21). Altfel spus, procesarea microinstrucțiunii curente s-a încheiat și SEQ revine în starea 0 pentru *fetch*-ul următoarei microinstrucțiuni.
- dacă $\overline{\text{MIR}}_{19} \cdot \overline{\text{MIR}}_{18} = 0$ înseamnă că microinstrucțiunea curentă lansează efectiv o operație cu memoria (are IFCH, READ sau WRITE codificat în câmpul OPERAȚII CU MEMORIA) În acest caz, din starea 1, automatul SEQ trece în starea 2 și mai apoi în starea 3 pentru a controla efectiv operația de transfer procesor-memorie:

IFCH –citire cod instrucțiune din memorie

READ –citire date (operand) din memorie

WRITE –scriere date (rezultat) în memorie.

În starea 2 este testat semnalul BUSY (blocul (8) din figura 4.21); BUSY este semnalul de stare aferent memoriei. Dacă $\text{BUSY}=1$ (memoria este ocupată, de exemplu cu un transfer DMA anterior declanșat sau cu un ciclu de regenerare), SEQ rămâne în starea 2 (bucla notată cu (9) în figura 4.21) așteptând dezactivarea semnalului BUSY. Când BUSY „cade” în 0 (când memoria devine liberă) automatul SEQ activează comanda MREQ (*Memory REQuest* = cerere de acces la memorie) (10) și trece în starea 3. Durata stării 2 depinde deci de starea memoriei. Dacă, la intrarea în starea 2, BUSY este găsit în 0, atunci SEQ stă în starea 2 o singură perioadă de tact.

În starea 3, SEQ menține activă cererea de acces la memorie și suplimentar activează comanda EN3 (blocul (11) din figura 4.21). Comanda EN3 (*enable* din starea 3)

va valida decodificatorul aferent câmpului MIR₁₉₊₁₈ (câmpul OPERAȚII CU MEMORIA). Acest decodificator va activa semnalele (comenziile):

- MEMR (MEMory Read) dacă în MIR₁₉₊₁₈ este codificată microcomanda IFCH sau microcomanda READ.
- MEMW (MEMory Write) dacă în MIR₁₉₊₁₈ este codificată microcomanda WRITE.

Comenziile MEMR și MEMW vor fi emise de către procesor (ca semnale de comandă pe bus) și vor fi recepționate la nivelul memoriei (pentru activarea operațiilor de citire din și respectiv de scriere în memorie). Acest decodificator (*hardware adițional*) simplifică sarcina SEQ pe durata stării 3 (a se compara organograma din figura 4.21 cu organograma din figura 3.7). În figura 4.22 este redată structura acestui *hardware adițional*.

În starea 3 SEQ testează din nou semnalul BUSY (blocul (12) din figura 4.21), așteptând astfel înceierea operației de transfer cu memoria (bucla notată cu (13) în organograma 4.21). Când BUSY “cade” în 0 (când ciclul de transfer cu memoria se încheie), SEQ mai menține activă doar comanda EN₃ și pe următorul impuls de tact, va reveni în starea 0 pentru *fetch*-ul următoarei microinstructiuni (tranziția notată cu (15) în organograma 4.21).

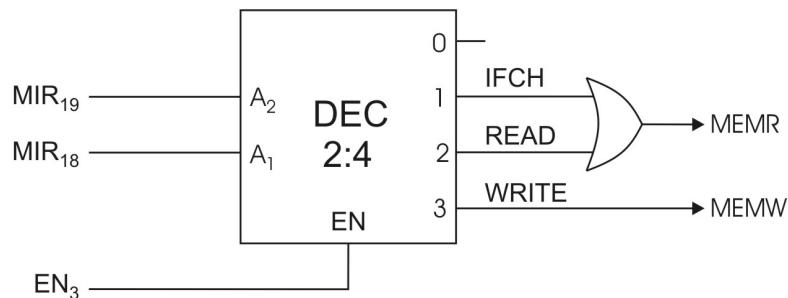


Fig 4.22 Generarea comenziilor MEMR și MEMW pentru memorie

La încheierea unei operații de citire din memorie (când automatul este în starea 3 și se dezactivează semnalul BUSY) va trebui activată și comanda de încărcare în procesor a informațiilor citite din memorie. Această comandă de încărcare se poate genera cu ajutorul unei scheme simple care implementează următoarele ecuații:

$$PmIR_3 = IFCH \cdot EN_3 \cdot \overline{BUSY} \quad (4.7)$$

$$PmMDR_3 = READ \cdot EN_3 \cdot \overline{BUSY} \quad (4.8)$$

Comanda PmIR₃ este comanda de încărcare în registrul IR a instrucțiunii extrase din memorie; se va activa dacă în microinstructiunea curentă este codificată microcomanda

IFCH. În cadrul microprogramului de emulare, această microinstructiune o vom regăsi în cadrul microrutinei **IF** (*Instruction Fetch*).

Comanda **PmMDR₃** este comanda de încărcare în registrul MDR a datelor extrase din memorie; se va activa dacă în microinstructiunea curentă este codificată microcomanda **READ**. În cadrul microprogramului de emulare, vom regăsi astfel de microinstructiuni în cadrul microrutinelor **Fetch Operand**. Dacă ar fi să detaliem puțin, în timpul procesării unui program pe procesorul nostru CISC, vom identifica următoarele tipuri de accese la date stocate în memorie:

- pentru citirea unui operand imediat (citirea unei constante); un astfel de acces vom identifica în microrutina **FOS** (*Fetch Operand Sursă*) aferentă modului de adresare imediat (fig 4.9). Reamintim faptul că adresarea imediata la destinație reprezintă cod ilegal.
- pentru citirea unui operand adresat indirect sau indexat; astfel de accese vom identifica în microrutinile **FOS** (*Fetch Operand Sursă*) și **FOD** (*Fetch Operand Destinație*) aferente modurilor de adresare indirect și indexat (fig 4.10 și 4.11).
- pentru citirea unui **INDEX**; astfel de accese vom identifica în microrutinile **FOS** (*Fetch Operand Sursă*) și **FOD** (*Fetch Operand Destinație*) aferente modului de adresare indexat (fig 4.11) precum și în microrutinile de execuție aferente instrucțiunilor **JMP**, **CALL** și de salt condiționat, cu mod de adresare indexat (fig 4.6.b.).
- pentru citirea unui **OFFSET**; astfel de accese vom identifica în microrutinile **EX** (de Execuție) aferente instrucțiunilor **JMP**, **CALL** și de salt condiționat, cu mod de adresare imediat (fig 4.6.a.).

Toate aceste accese vor fi inițiate prin microinstructiuni care au codificată microcomanda **READ** în câmpul **OPERAȚII CU MEMORIA** și toate aceste microinstructiuni, la finele stării 3, vor provoca activarea comenzi **PmMDR₃** (ecuația 4.8).

Să observăm că și în câmpul **DESTINAȚIE RBUS** al microinstructiunii generale regăsim o microcomandă **PmMDR** (fig 4.18). Această microcomandă se va activa în starea 1 a automatului **SEQ** și va fi utilizată pentru implementarea transferurilor registru-registrul (transferurile interne din cadrul procesorului, atunci când destinația codificată în microinstructiune este registrul MDR). Comanda globală de încărcare paralelă a registrului MDR se va obține prin reunirea celor două comenzi distințe (ecuația 4.9).

$$\text{PmMDR}_{\text{global}} = \text{PmMDR} + \text{PmMDR}_3 \quad (4.9)$$

Pentru implementare, ecuația (4.9) revendică o poartă SAU cu 2 intrări.

B. Graful de tranziții al microsevențiatorului

Pornind de la organograma automatului **SEQ** (figura 4.21) se poate elabora graful de tranziții al acestuia (fig 4.23). În graful de tranziții sunt reprezentate cele patru stări din organigramă și tranzițiile dintre stări, figurate prin arce orientate. Cele 4 stări ale automatului pot fi codificate cu ajutorul a 2 biți de stare și codificarea pe care o propunem

este evidențiată atât în organigramă cât și în graful de tranziții. Trebuie subliniat faptul că stările au fost codificate adiacent (două stări interconectate printr-un arc de tranziție diferă între ele printr-un singur bit de stare). Regula codificării adiacente a fost respectată pe toate arcele din graf deoarece, la tranziția dintre două stări codificate neadiacent, se pot genera impulsuri parazite pe ieșirile automatului (comenzi parazite). Codificarea adiacentă elimină comenzile parazite.

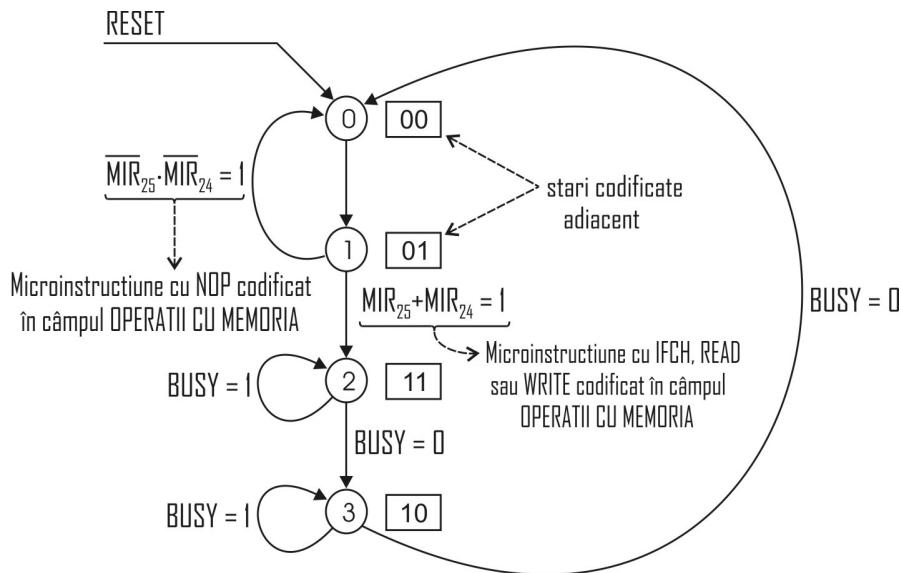


Fig 4.23 Graful de tranziții al automatului SEQ aferent procesorului CISC

Pe arcele care reprezintă tranziții condiționate sunt înscrise și condițiile care validează tranzițiile respective.

Exemple:

- Tranziția din starea 2 în starea 3 se poate face numai dacă $BUSY=0$ (memoria nu este ocupată cu un ciclu de citire sau de scriere anterior startat sau cu un ciclu de regenerare). Această condiție rezultă evident din organograma automatului (figura 4.21).
- Tranziția (revenirea) din starea 1 în starea 0 se realizează numai dacă: $\overline{MIR}_{19} \bullet \overline{MIR}_{18} = 1$ (dacă microinstructiunea curentă are codificat NOP în câmpul OPERAȚII CU MEMORIA)

Să menționăm că organograma (figura 4.21) și graful de tranziții (figura 4.23) sunt necesare în fază de proiectare a automatului SEQ.

C. Schema bloc a microsevențiatorului

Schema bloc a automatului SEQ se obține pornind de la organograma automatului din figura 4.21. Variabilele testate în organigramă vor fi intrări în automat, iar comenzi activate în organigramă (în diferitele stări) vor fi ieșiri din automat. Automatul SEQ este un automat secvențial sincron; trecerea din starea curentă în starea următoare se face sub controlul semnalului de tact al procesorului (CLK_P). Inițializarea automatului se face cu ajutorul semnalului **RESET** care realizează de fapt inițializarea procesorului și respectiv calculatorului (în ansamblu). Schema bloc a automatului SEQ este redată în figura 4.24.

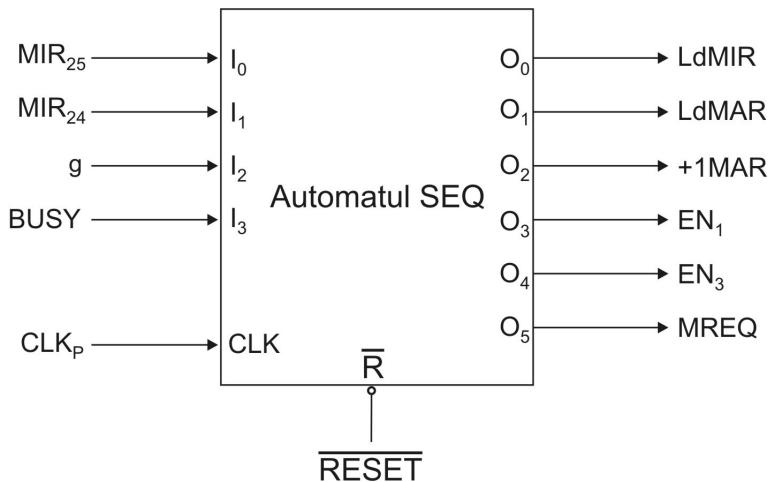


Fig 4.24 Schema bloc a automatului SEQ

Există mai multe soluții de implementare a automatului SEQ în *hardware*. La 3.5.4, 3.5.6 și 3.5.7 au fost prezentate trei exerciții (trei soluții) de implementare în *hardware* a automatului SEQ aferent procesorului didactic. Lăsăm în seama cititorului efectuarea unor exerciții similare de proiectare pentru automatul SEQ aferent procesorului CISC. Comparativ, organograma automatului SEQ aferent procesorului CISC este mai simplă. Prin urmare, implementările ce vor rezulta vor fi și ele mai simple.

4.2.5 Schema bloc a unității de control microprogramate

Un program este format dintr-o succesiune de instrucțiuni. Unitatea de control a procesorului, denumită și bloc de generare a comenzi (BGC), generează toate semnalele de comandă (microcomenzi) prin care se controlează procesarea instrucțiunilor (*fetch*-ul și execuția acestora). Structura de principiu a unității de control microprogramate a fost prezentată la 3.2 (fig 3.1), particularizată pentru procesorul didactic la 3.4.1 (fig 3.4) și detaliată la 3.6 (fig 3.31). Particularizarea/detalierea a ținut cont de succesorii și de indecșii definiți în formatul microinstrucțiunii de ramificație aferentă procesorului didactic.

Schema de principiu din figura 3.1 poate fi particularizată/detaliată și pentru procesorul CISC. Această particularizare trebuie să țină cont de formatul microinstrucțiunii generale (fig. 4.18), de condițiile testate (tabelul 4.6) și de indecșii definiți în cadrul microinstrucțiunii generale (tabelul 4.7). Particularizarea pentru procesorul CISC este redată în figura 4.25.

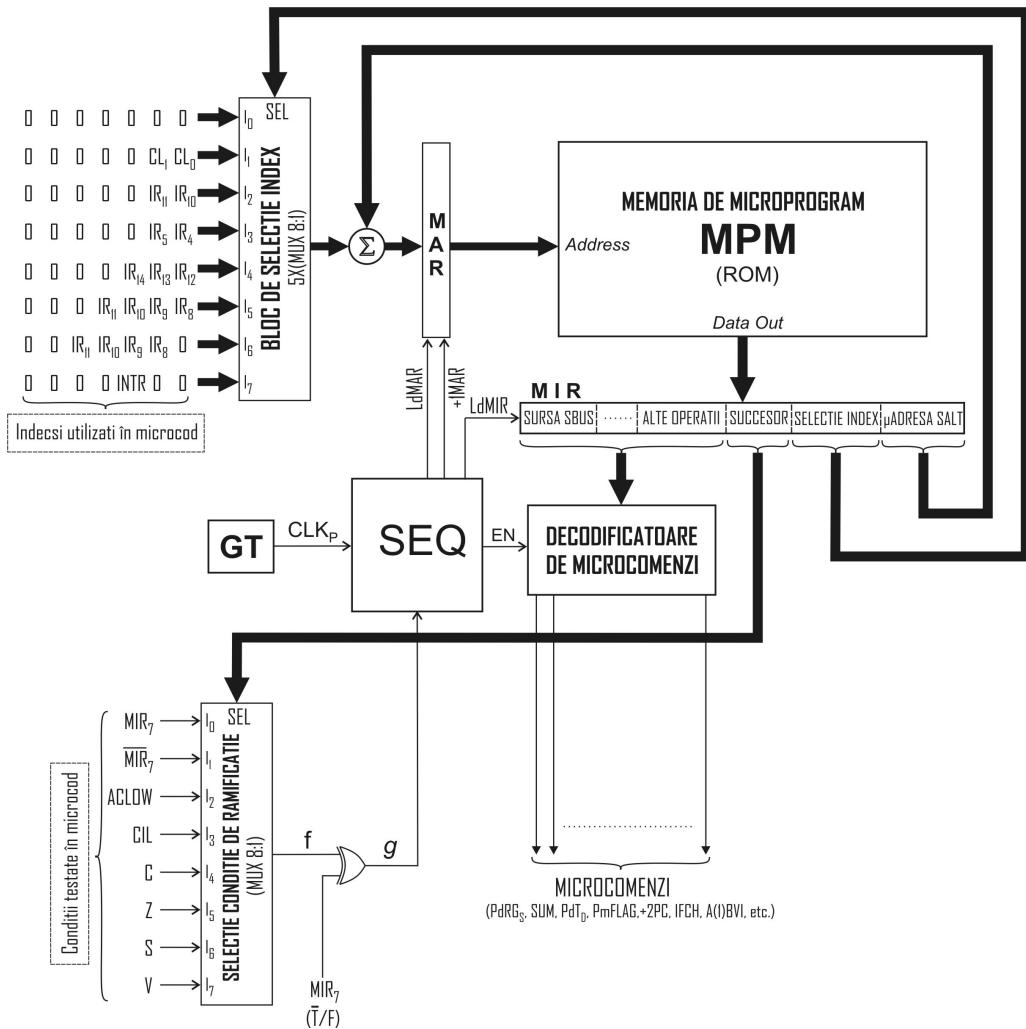


FIG 4.25 Schema bloc a unității de control microprogramate aferentă procesorului CISC

În structura unității de control din figura 4.25 regăsim următoarele blocuri componenete:

- MIR** – *MicroInstruction Register* (registrul microinstructiunii). În MIR se încarcă microinstructiunea selectată în MPM de microadresa prezentă în MAR (*fetch*-ul microinstructiunii). Pentru încărcarea microinstructiunii în MIR, în starea 0, automatul SEQ activează comanda **LdMIR** (figura 4.21).
- MAR** – *MicroAddress Register* (registrul de microadrese). MAR joacă rol de *microprogram counter* (adreseză în MPM microinstructiunea ce urmează a fi citită și încărcată în MIR (*fetch*-ul microinstructiunii)). Pentru încărcarea în MAR a microadresei următoarei microinstructiuni, SEQ utilizează două comenzi: **LdMAR** (pentru succesorii JUMP și JUMPI) și respectiv **+1MAR** (pentru succesorul STEP).
- MPM** – *MicroProgram Memory* (memoria de microprogram). MPM conține microprogramul de emulare a setului de instrucțiuni aferent procesorului CISC. Microprogramul este format dintr-o colecție de microrutine: microrutina *fetch* instrucțiune, microrutine *fetch* operand sursă (câte una pentru fiecare mod de adresare), microrutine *fetch* operand destinație (câte una pentru fiecare mod de adresare), microrutine de execuție (câte una pentru fiecare instrucțiune) și microrutina de întreprere (vezi organizarea microprogramului sintetizată în figura 4.17).
- SEQ** – *SEQuencer* (microsecvențiator). Automatul SEQ generează comenzi interne prin care controlează *fetch*-ul și execuția microinstructiunilor; a fost prezentat la 4.2.4.

SELECTIE CONDIȚIE DE RAMIFICAȚIE – bloc *hardware* care generează funcția globală de ramificație **g**. Acest bloc are rol în execuția succesorilor (vezi explicațiile relative la blocurile (3), (4) și (5) din figura 4.21)

Blocul poate fi implementat cu un multiplexor 8:1 la care se adaugă o poartă SAU-EXCLUSIV. Pe intrările multiplexorului se aplică, condițiile care vor trebui testate pe parcursul microprogramului de emulare. Condițiile au fost preluate din tabelele (4.6) și respectiv (4.8). Cele două tabele detaliază câmpul **SUCCESOR** codificat în microinstructiunea generală (fig. 4.18). În aceste condiții, multiplexorul va genera funcția **f** (condiția ce va fi testată) iar poarta SAU-EXCLUSIV va genera funcția globală de ramificație (**g**).

Funcția **g** va fi testată de către automatul SEQ, (figura 4.21) pentru fiecare microinstructiune procesată, în starea 1. Dacă **g=1**, atunci SEQ va genera comanda **LdMAR** (se execută succesorul JUMPI (respectiv JUMP dacă indexul selectat va fi $\text{INDEX}_0=0$). Asta se întâmplă când condiția testată în microinstructiune (pe fals sau pe adevărat, după cum MIR_7 are valoarea 1 sau 0) este îndeplinită. Dacă **g=0**, atunci SEQ va genera comanda **+1MAR** (se execută succesorul STEP). Asta se întâmplă când condiția testată în microinstructiune (pe fals sau pe adevărat, după cum MIR_7 are valoarea 1 sau 0) este neîndeplinită.

BLOC DE SELECTIE INDEX –este format din 5 multiplexoare 8:1 și generează cei 8 indecsi definiți la 4.2.2 (tabelul 4.7). Are deci rol în execuția succesorilor JUMPI și respectiv JUMP. Biții MIR₁₀₋₈ (câmpul SELECTIE INDEX din microinstrucțiunea curentă) se aplică pe intrările de selecție aferente celor 5 multiplexoare, selectând astfel indexul specificat în cadrul microinstrucțiunii. Dacă indexul selectat este diferit de zero (INDEX₁=INDEX₇) atunci se va executa într-adevăr un successor JUMPI (vezi ecuația (4.5) de la 4.2.4.A). Dacă indexul selectat este INDEX₀=0 atunci successorul devine JUMP (vezi ecuația (4.6) de la 4.2.4.A).

Observație: *Indecșii sunt codificați pe 7 biți (vezi tabelul 4.7) deoarece câmpul MICROADRESA DE SALT este pe 7 biți (fig 4.18). Cei mai semnificativi 2 biți din index (biții IND₆ și IND₅ din tabelul 4.7) au valoarea zero, în toți cei 8 indecsi definiți. Acești 2 biți nu vor necesita multiplexoare pentru selectia lor deoarece pot fi generați foarte simplu ca și constante 0 (prin conectare la "0" a linilor respective). Prin urmare, doar cei mai puțin semnificativi 5 biți din index (biții IND₄-IND₀ din tabelul 4.7) revendică multiplexoare 8:1 pentru selecție. Orice idee care conduce la simplificări în hardware este binevenită !*

- Σ –sumator pe 7 biți care generează microadresele de salt (pentru succesorii JUMPI și respectiv JUMP). În cazul succesorilor JUMPI se face întradevăr o operație de adunare (operația (4.5) de la 4.2.4.A). În cazul succesorilor JUMP se face o adunare cu INDEX₀=0 (operația (4.6) de la 4.2.4.A). Prin acest artificiu de adunare cu constanta 0, succesorii JUMPI (tabelul 4.6) se transformă în succesorii JUMP echivalenți (tabelul 4.7).

4.2.6 Proiectarea blocurilor din componența unității de control microprogramate

În paragraful anterior au fost introdusă schema bloc a unității de control microprogramate (figura 4.25) și au fost descris succint rolul fiecărui bloc din componența acestei unități. În cadrul prezentului paragraf vom trece de la descrierea succintă la proiectarea detaliată a fiecărui bloc din componența unității de control. Remintim faptul că proiectarea automatului SEQ a fost deja realizată (paragraful 4.2.4). Vom continua, prin urmare, cu celelalte blocuri din structura unității microprogramate aferente procesorului CISC.

A. Blocul de decodificare a microcomenzilor

Microcomenzile se generează prin decodificarea câmpurilor SURSA SBUS, SURSA DBUS, OPERAȚIE ALU, DESTINAȚIE RBUS, OPERAȚII CU MEMORIA și ALTE OPERAȚII din componența microinstrucțiunii generale (fig 4.18).

Câmpul OPERAȚII CU MEMORIA este un câmp de 2 biți și comenzile codificate în acest câmp sunt comenzi pentru memorie. Transferurile procesor-memorie sunt controlate de către automatul SEQ, în starea 3 (vezi organograma SEQ din figura 4.21). Prin urmare, decodificatorul care generează comenzile pentru memorie trebuie validat cu semnalul de validare EN₃. Acest decodificator a fost implementat și explicat la 4.2.4.A (fig 4.22).

Câmpurile **SURSA SBUS**, **SURSA DBUS**, **OPERAȚIE ALU** și **DESTINAȚIE RBUS** sunt câmpuri de 4 biți. Cu ajutorul microcomenzilor codificate în aceste patru câmpuri se controlează transferurile registru-registru (transferurile interne). Informația transferată traversează întotdeauna unitatea ALU și la nevoie, poate fi procesată în ALU. În aceste transferuri pot fi specificate:

- două registre sursă (unul emis pe SBUS și celălalt emis pe DBUS),
- o operație ALU (care se va efectua între operanții preluati de pe SBUS și respectiv DBUS)
- un registru destinație (în care se va încărca rezultatul obținut pe ieșirile ALU).

Toate operațiile aritmetico-logice cu 2 operanzi se efectuează în această manieră (într-o singură perioadă de tact procesor).

Există desigur și operații aritmetico-logice, cum ar fi de exemplu deplasările și rotirile, care presupun procesarea unui singur operand și există și posibilitatea efectuării unui transfer simplu registru-registru, fără procesarea în ALU a datelor transferate.

Toate aceste operații de transfer se desfășoară pe durata stării 1 a automatului SEQ. Prin urmare, decodificatoarele aferente acestor câmpuri trebuie validate cu semnalul de validare **EN₁**, semnal activat de către automatul SEQ pe durata stării 1 (vezi organograma SEQ din figura 4.21). Decodificatoarele aferente câmpurilor **SURSA SBUS**, **SURSA DBUS**, **OPERAȚIE ALU** și **DESTINAȚIE RBUS** sunt prezentate în figura 4.26

Decodificatorul notat cu 1 decodifică câmpul **SURSA SBUS** (**MIR₃₅₊₃₂**). Pe ieșirile decodificatorului se obțin microcomenzi (de tip nivel) care emit diverse surse pe SBUS, specificate în ordinea în care au fost codificate în formatul microinstrucțiunii generale (fig 4.18). Microcomanda **NONE** înseamnă „nici-o sursă emisă pe SBUS”. Următoarele 10 microcomenzi (**PdFLAGS**, **PdRGs**, **PdSPs**, **PdT_s**, **PdPCs**, **PdIVRs**, **PdADRs**, **PdMDRs**, **PdIR[7+0]s**) acționează asupra regiszrelor conectate la SBUS (fig 4.1). Fiecare dintre cele 10 microcomenzi va emite pe SBUS datele existente în regiszrul specificat. Indicele „s” care apare în denumirile acestor microcomenzi specifică faptul că regiszrul vizat de microcomandă va fi emis pe SBUS. Este necesară această precizare deoarece toate regiszrele din structura procesorului CISC (fig 4.1) sunt conectate atât la SBUS cât și la DBUS.

În figura 4.27 este evidențiat modul în care sunt conectate regiszrele T și PC la SBUS. Microcomenziile evidențiate în figura 4.27 produc următoarele acțiuni:

PdT_s, -emite conținutul regiszrului T pe SBUS

Pd \bar{T} _s, -emite conținutul regiszrului T, inversat bit cu bit, pe SBUS

PdPCs, -emite conținutul regiszrului PC pe SBUS

Celelalte regiszre din structura procesorului sunt conectate la SBUS prin scheme similare. Astă înseamnă că, pe cele 16 linii ale SBUS-ului sunt implementate funcții SAU-CABLAT. Prin urmare, dacă în câmpul **SURSA SBUS** din microinstrucțiunea curentă este codificată microcomanda **NONE** atunci liniile SBUS-ului vor fi în stare de înaltă impedanță (nu se emite nici-o sursă pe SBUS).

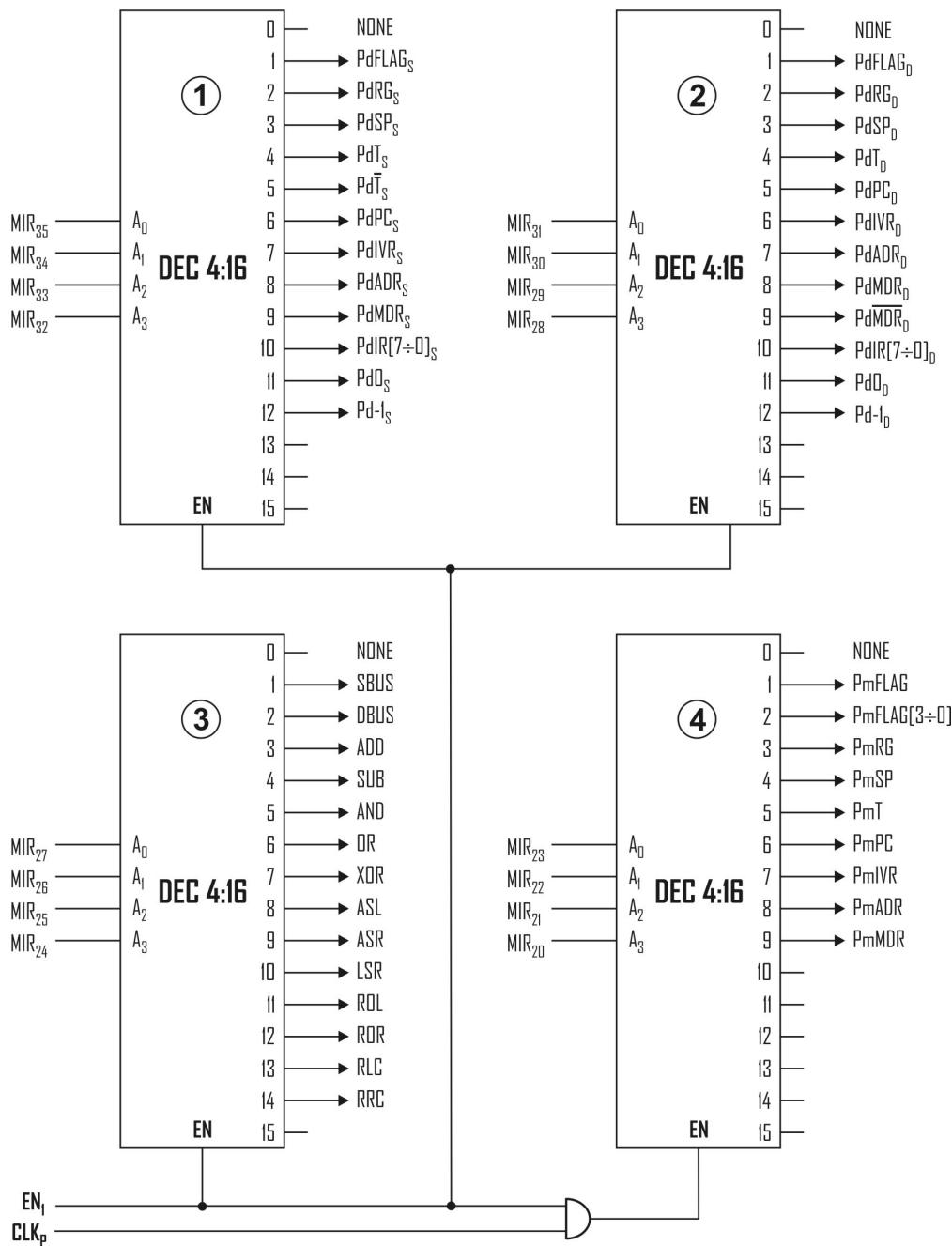


Fig 4.26 Decodificatoarele de microcomenzi aferente câmpurilor SURSA SBUS, SURSA DBUS, OPERAȚIE ALU și DESTINAȚIE RBUS

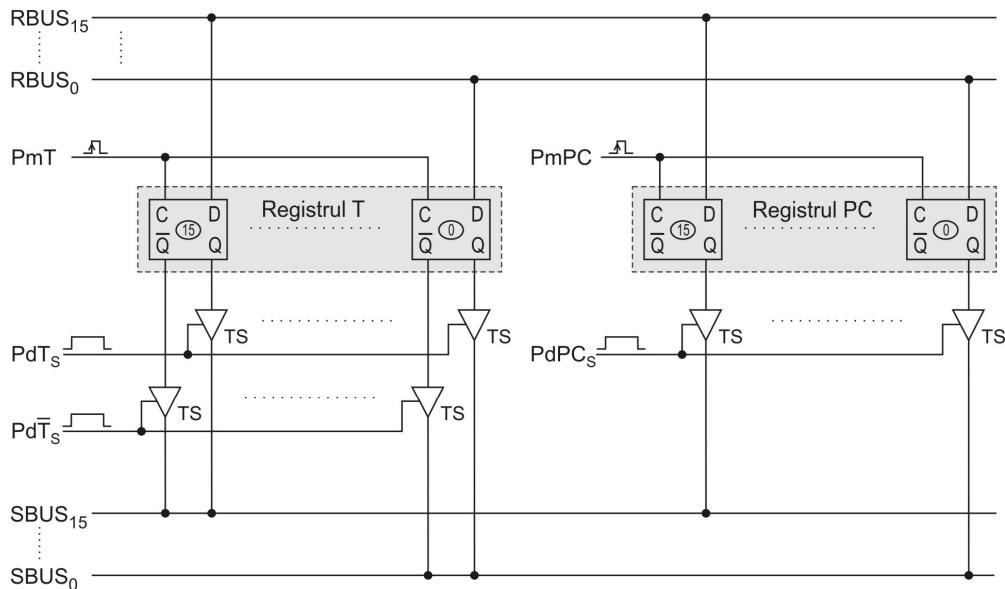


Fig 4.27 Conectarea registrelor T și PC la SBUS

In figura 4.28 este ilustrat modul în care sunt generate constantele 0 și -1 pe SBUS.

Decodificatorul notat cu 2 în figura 4.26 decodifică câmpul **SURSA DBUS** (MIR_{31-28}). Pe ieșirile decodificatorului se obțin microcomenzi (de tip nivel) care emit diversele surse pe DBUS, specificate în ordinea în care au fost codificate în formatul microinstrucțiunii generale (fig. 4.18). Indicele „D” care apare în denumirile acestor microcomenzi specifică faptul că registrul vizat de microcomandă va fi emis pe DBUS (nu pe SBUS!). Implementarea DBUS-ului este similară cu cea a SBUS-ului (funcție SAU-CABLAT pe fiecare dintre cele 16 linii de bus).

Decodificatorul notat cu 3 în figura 4.26 decodifică câmpul **OPERAȚIE ALU** (MIR_{27-24}). Pe ieșirile decodificatorului se obțin microcomenzi (de tip nivel) dedicate unității ALU, specificate în ordinea în care au fost codificate în formatul microinstrucțiunii generale (fig. 4.18). Să observăm că microcomenziile de deplasare și rotere le-am codificat în câmpul **OPERAȚIE ALU**. Prin urmare, operațiile de deplasare și rotere vor fi implementate în ALU (vezi cele 2 observații de la 2.7.2.D).

Decodificatorul notat cu 4 în figura 4.26 decodifică câmpul **DESTINAȚIE RBUS** (MIR_{23-20}). Pe ieșirile decodificatorului se obțin microcomenzi „primește registru”, specificate în ordinea în care au fost codificate în formatul microinstrucțiunii generale (fig. 4.18). Aceste microcomenzi trebuie să fie de tip impuls și din acest motiv decodificatorul este validat cu semnalul $EN=EN_1 \cdot CLK_P$ (vezi figurile 2.18 și 3.34 cu explicațiile aferente).

Microcomenziile „primește registru” încarcă datele prezente pe RBUS, în registre. În cazul procesorului CISC (fig. 4.1), pe RBUS vom avea întotdeauna datele obținute pe ieșirea ALU, deoarece ALU este singura sursă conectată la SBUS.

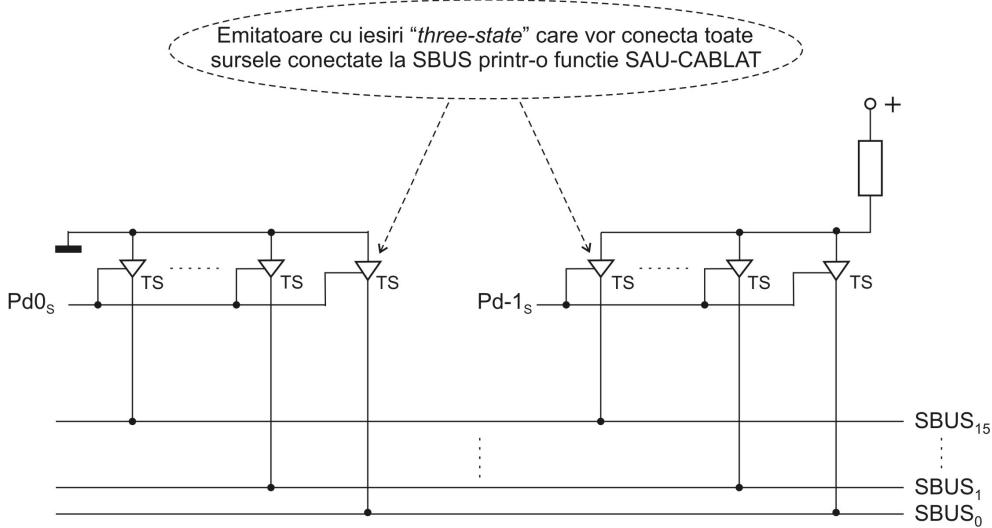


Fig 4.28 Generarea constantelor 0 și -1 pe SBUS

În figura 4.27 este ilustrată acțiunea comenziilor PmT și PmPC iar în figura 4.20 este ilustrată acțiunea comenzi PmRG; o astfel de microcomandă „primește regiszru” vom regăsi la nivelul fiecărui regiszru din structura procesorului.

Comenzi codificate în câmpul ALTE OPERAȚII (MIR₁₇₊₁₄) al microinstrucțiunii generale din figura 4.18, trebuie executate tot în starea 1 a automatului SEQ. Validarea decodificatorului aferent acestor microcomenzi (fig 4.29) se va face deci tot cu semnalul EN₁, iar microcomenzi sunt asignate ieșirilor decodificatorului 4:16 în ordinea în care au fost codificate în formatul microinstrucțiunii generale (fig 4.18):

- NONE –nici-o comandă
- +2SP –incrementare cu 2 a *stack pointer*-ului (necesară la operația *pop*)
- 2SP –decrementare cu 2 a *stack pointer*-ului (necesară la operația *push*)
- +2PC –incrementare cu 2 a *program counter*-ului (în microrutina *Instruction Fetch*)
- A(1)BE0 –setarea bistabilului de excepție numărul 0 (asociat excepției ACLOW)
- A(1)BE1 –setarea bistabilului de excepție numărul 1 (asociat excepției CIL)
- PdCOND_A –setarea *flag*-urilor C, Z, S și V în acord cu rezultatul obținut pe ieșirea ALU (vezi observația nr. 4 de la 4.2.3)
- C_{IN}, PdCOND_A –comandă compusă care, pe lângă setarea *flag*-urilor, aplică "1" pe intrarea C_{IN} a unității ALU (pentru operația de incrementare cu 1)
- PdCOND_L –setarea *flag*-urilor Z și S în acord cu rezultatul obținut pe ieșirea ALU (vezi observația nr. 4 de la 4.2.3)

- A(1)BVI –setarea bistabilului de validare a întreruperilor (la instrucțiunea EI)
- A(0)BVI –resetarea bistabilului de validare a întreruperilor (la instrucțiunea DI)
- A(0)BPO –resetarea bistabilului de pornire-oprire (instrucțiunea HALT, fig 2.64)
- INTA, -2SP –achitarea întreruperii (*INTA=INTerrupt Acknowledge*) și decrementarea cu 2 a *stack pointer*-ului (va fi utilizată în microrutina de întrerupere)
- A(0)BE, A(0)BI –resetarea bistabililor de excepție și respectiv de întrerupere. Se face în microrutina de întrerupere, după generarea vectorului de întrerupere/excepție. După generarea vectorului trebuie resetat bistabilul în care s-a memorat întrerupere/excepția, altfel se ajunge la o tratare recursivă a aceleiași întreruperi/excepții.

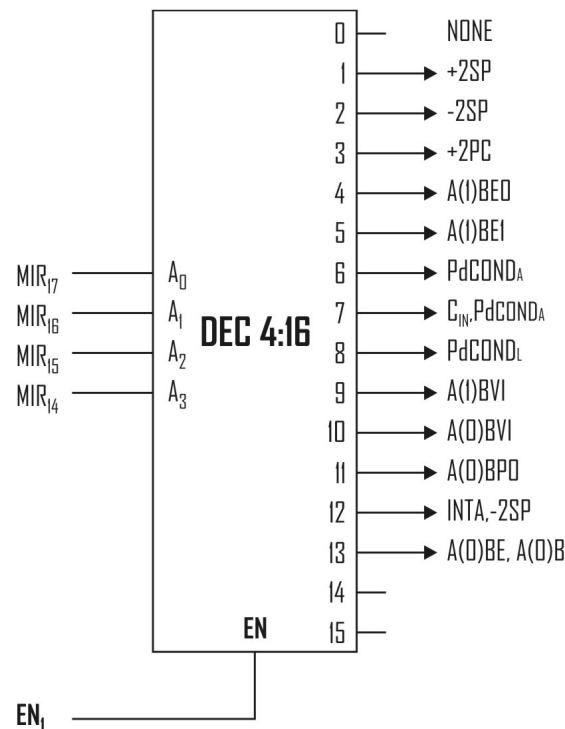


Fig 4.29 Decodificatorul de microcomenzi aferent câmpului ALTE OPERAȚII

B. Blocul de selecție index

În baza unui raționament similar celui aplicat la procesorul didactic, blocul de selecție index poate fi implementat în *hardware* cu ajutorul unui set de multiplexoare (fig 4.30).

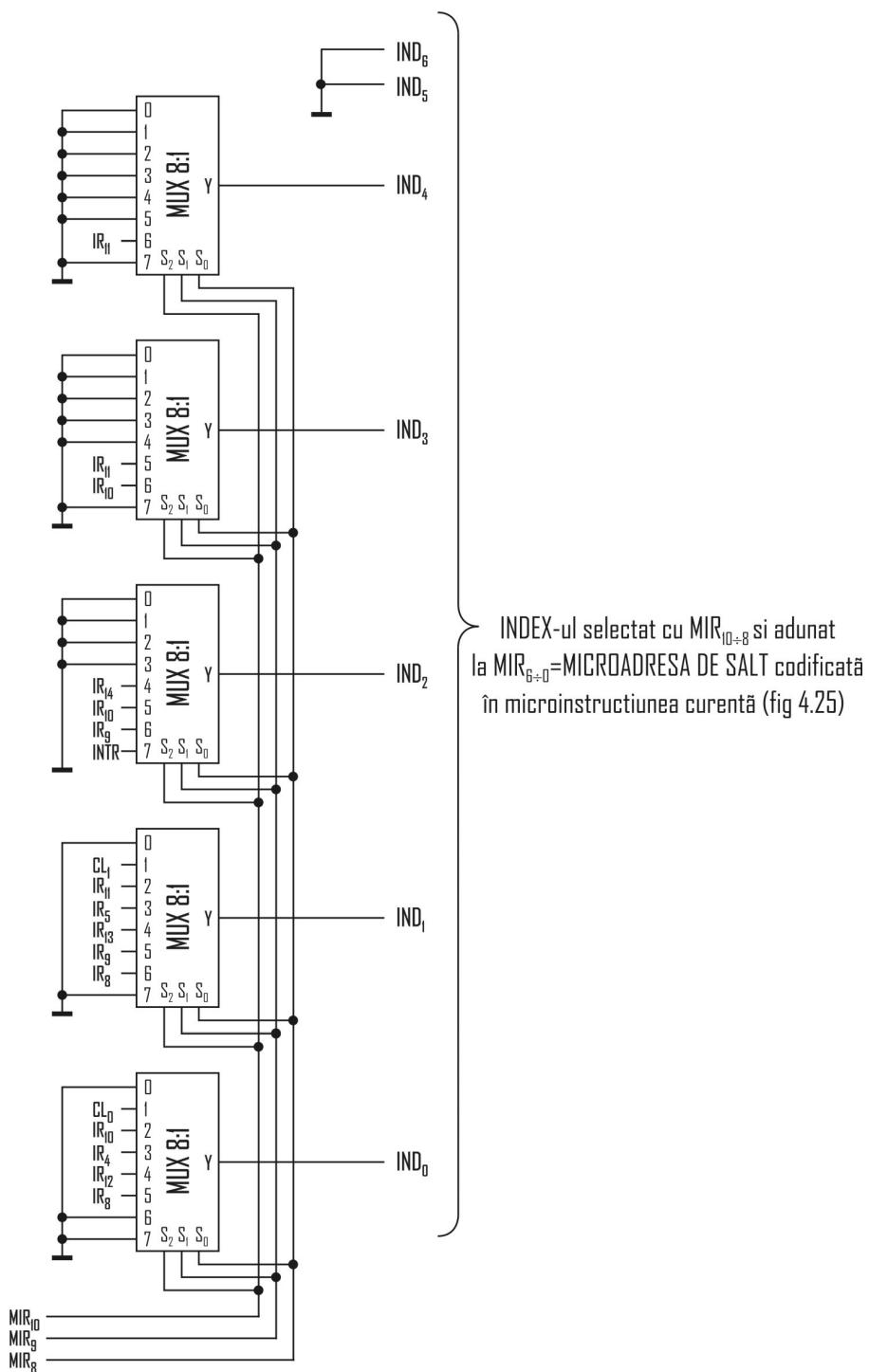


Fig 4.30 Blocul de selecție index aferent procesorului CISC

Din schema principală de organizare a microprogramului de emulare (fig 4.17) au rezultat indecșii care vor fi utilizati pentru implementarea ramificațiilor indexate din cadrul microprogramului. Acești indecși au fost sintetizati în tabelul 4.7. Tabelul 4.7 conține 8 indecși; primul index are valoarea zero iar următorii 7 sunt diferiți de zero. Reamintim faptul că primul index ($\text{INDEX}_0=0$) permite transformarea succesorilor JUMPI (tabelul 4.6) în succesori JUMP echivalenți (tabelul 4.8).

Strategia de proiectare aplicată în cazul blocului de selecție index aferent procesorului didactic (paragraful 3.7.2) va fi aplicată și aici. Diferențele care apar sunt:

- La procesorul didactic au fost definiți doar 3 indexi (tabelul 3.3) ceea ce a condus la un câmp **SELECTIE INDEX** de 2 biți în formatul microinstrucțiunii (fig 3.2) și respectiv la o implementare bazată pe multiplexoare 4:1 (vezi fig 3.37).
- La procesorul CISC au fost definiți 8 indexi (tabelul 4.7) ceea ce a condus la un câmp **SELECTIE INDEX** de 3 biți în formatul microinstrucțiunii (fig 4.18) și respectiv la o implementare bazată pe multiplexoare 8:1 (fig 4.30).

Cei 3 biți din componența câmpului **SELECTIE INDEX** aferent microinstrucțiunii curente (MIR_{10+8}) sunt aplicați pe intrările de selecție (S_{2+0}) aferente celor 5 multiplexoare 8:1 din figura 4.30. Intrările celor 5 multiplexoare au fost stabilite pe baza tabelului 4.7. De exemplu, dacă pe intrările de selecție se aplică codul $\text{MIR}_{10+8}=001$ atunci, la nivelul fiecărui multiplexor, va fi selectată intrarea 1 și valoarea existentă pe această intrare va fi transferată la ieșirea **Y** a multiplexorului. Prin urmare, dacă $\text{MIR}_{10+8}=001$, atunci pe ieșirile celor 5 multiplexoare se obține $\text{IND}_{4+0}=0, 0, 0, 0, \text{CL}_1, \text{CL}_0$. Având în vedere că biții IND_6 și IND_5 sunt întotdeauna zero, rezultă $\text{IND}_{6+0}=0, 0, 0, 0, 0, \text{CL}_1, \text{CL}_0$. În concluzie, dacă $\text{MIR}_{10+8}=001$, pe ieșirile blocului de selecție index din figura 4.30 se obține $\text{IND}_{6+0}=\text{IND}_{\text{INDEX}}_1=0, 0, 0, 0, 0, \text{CL}_1, \text{CL}_0$.

În mod analog, cu ajutorul câmpului **MIR₁₀₊₈** care poate lua 8 valori distincte (de la 000₂ și până la 111₂), sunt selectați și ceilalți indecși definiți în tabelul 4.7.

C. Blocul de generare a funcției globale de ramificație

Funcția globală de ramificație (**g**) este testată de automatul SEQ (fig 4.25), în starea 1 (fig 4.21) pentru a decide succesorul ce va fi executat:

- dacă **g=1** atunci SEQ va activa comanda LdMAR (fig 4.25). Se va executa deci succesorul JUMPI sau JUMP. Cu alte cuvinte, în microcod, se va executa un salt indexat sau un salt simplu. Reamintim faptul că succesorul JUMP reprezintă un caz particular al succesorului JUMPI (dacă $\text{MIR}_{10+8}=000$, atunci INDEX-ul selectat va avea valoarea zero (tabelul 4.7) și JUMPI devine JUMP).
- dacă **g=0** atunci SEQ va activa comanda +1MAR (se va executa succesorul STEP). Cu alte cuvinte, în microcod, se va trece secvențial la următoarea microinstrucțiune.

Funcția globală de ramificație aferentă procesorului CISC va fi implementată (fig 4.31) analog cu cea aferentă procesorului didactic (fig 3.38).

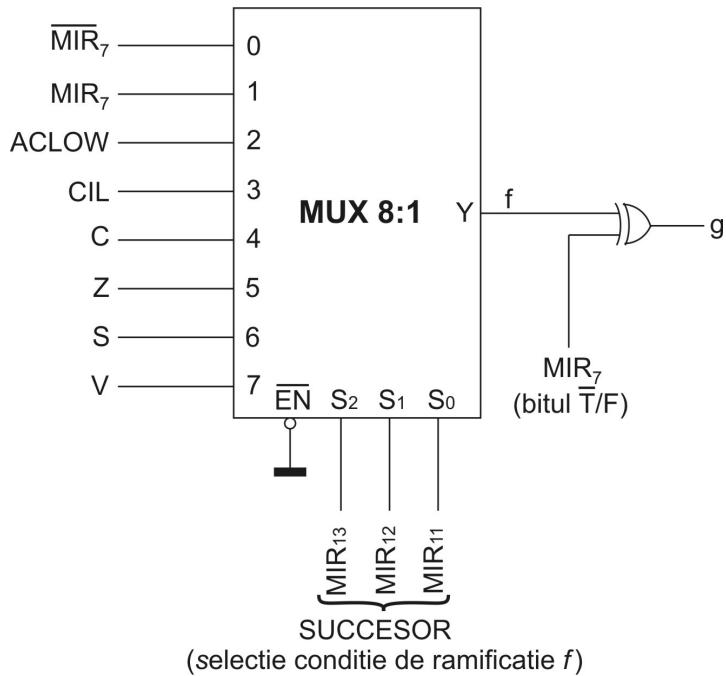


Fig 4.31 Generarea funcției globale de ramificație la procesorul CISC

La procesorul didactic a fost necesar un multiplexor 16:1 (fig 3.38) pentru a selecta condițiile cuprinse în tabelele 3.1 și respectiv 3.2, pe ieșirea multiplexorului obținându-se funcția testată (**f**). Pe baza funcției **f** și a bitului MIR_{12} ($\overline{T/F}$) s-a generat funcția globală de ramificație $g=f \oplus MIR_{12}$ (vezi paragraful 3.4.1). De altfel, tabelele 3.1 și 3.2 sunt identice din punctul de vedere al condițiilor testate; diferența între cele două tabele o face INDEX-ul (care are valoarea zero în cazul tabelului 3.1 și respectiv diferită de zero în cazul tabelului 3.2).

La procesorul CISC este suficient un multiplexor 8:1 (fig 4.31) pentru a selecta condițiile cuprinse în tabelele 4.6 și respectiv 4.8, pe ieșirea multiplexorului obținându-se funcția testată (**f**). Pe baza funcției **f** și a bitului MIR_7 ($\overline{T/F}$) se generează funcția globală de ramificație $g=f \oplus MIR_7$. Similar (cu situația prezentată la procesorul didactic), tabelele 4.6 și respectiv 4.8 sunt identice din punctul de vedere al condițiilor testate; diferența între cele două tabele o face INDEX-ul (care are valoarea zero în cazul tabelului 4.8 și respectiv diferită de zero în cazul tabelului 4.6).

Blocul de generare a funcției **g** aferentă procesorului didactic este activat numai dacă în registrul MIR se află o microinstrucțiune de ramificație (dacă microinstrucțiunea curentă este de ramificație). Din acest motiv, multiplexorul din figura 3.38 este validat cu $MIR_{19}=0$.

La procesorul CISC operăm cu un singur tip de microinstrucțiune (microinstrucțiunea generală) care conține întotdeauna și partea de ramificație (fig 4.18). Din acest motiv, multiplexorul din figura 4.31 este validat permanent.

4.2.7 Microprogramul de emulare

Microprogramul de emulare este compus dintr-o colecție de microrutine. Legăturile dintre aceste microrutine se stabilesc dinamic, pe parcursul procesării (emulării) instrucțiunilor programului aflat în execuție. Organizarea microprogramului de emulare a fost prezentată deja în figura 4.17, unde microrutinile au fost reprezentate (simbolic) prin segmente de dreapta iar pentru legăturile dintre microrutine au fost propuse puncte de ramificație implementate prin salturi indexate. Un număr de 7 indecs sunt evidențiați în figura 4.17, pentru implementarea legăturilor dintre microrutine. La acești 7 indecs, diferenți de zero, am adăugat indexul INDEX₀ care are valoarea zero și care are rolul de a transforma succesorii JUMPI (tabelul 4.6) în succesori JUMP (tabelul 4.8). Toți acești indecs sunt preluati și implementați și în formatul microinstrucțiunii generale (vezi fig 4.18 și tabelul 4.7). Să remarcăm faptul că inclusiv pentru implementarea punctului interruptibil se propune un salt indexat, cu INDEX₇=0, 0, 0, INTR, 0, 0.

Implementarea ramificațiilor (din cadrul microcodului) prin salturi indexate, vizează îmbunătățirea performanțelor procesorului CISC; legăturile (ramificațiile) implementate prin salturi indexate sunt mai rapide decât cele implementate prin salturi condiționate (vezi paragraful 3.11).

Introducerea microinstrucțiunii generale (paragraful 4.2.2) vizează tot îmbunătățirea performanțelor procesorului CISC; microinstrucțiunea generală ne va permite proiectarea unor microrutine foarte scurte.

Minimizarea lungimii microrutinelor și minimizarea timpilor de legătură dintre microrutine reprezintă cele două tehnici (de bază) pe care le vom aplica pentru optimizarea microprogramului de emulare, în ultimă instanță, pentru maximizarea vitezei de execuție (de emulare) a instrucțiunilor.

Microprogramul de emulare aferent procesorului CISC este redat în tabelul 4.9. Acest microprogram a rezultat printr-o transpunere în microcod a schemei de principiu din figura 4.17. Cu alte cuvinte, pentru obținerea microprogramului:

- au fost proiectate toate microrutinile (evidențiate simbolic prin segmente în figura 4.17)
- au fost implementate (prin salturi indexate) toate legăturile dintre microrutine (conform schemei de legături din figura 4.17)

În cadrul microcodului apar și alte puncte de ramificație implementate prin salturi condiționate (tabelul 4.8) sau chiar prin salturi indexate condiționate (tabelul 4.6). În mod cert, astfel de ramificații condiționate apar în microrutinile de execuție aferente instrucțiunilor de salt condiționat (vezi etichetele BEQ, BNE, BMI, BPL, BCS, BCC, BVS și BVC); în aceste microrutine sunt testate desigur *flag*-urile de condiții: C, Z, S, V.

Pentru specificarea microinstrucțiunilor am utilizat limbajul mnemonic introdus în capitolul 3 (vezi exemplele și explicațiile de la 3.8.2.A, 3.8.2.B și 3.8.2.C). Singura diferență care apare, rezultă firesc din capacitatele sporite ale microinstrucțiunii generale, care permite codificarea unui set amplu de operații concurente (vezi 4.2.2).

Tabelul 4.9 Microprogramul de emulare:

Eticheta NPM	Adresa NPM	Micrinstrucțiunea	Comentarii
IFC:	00H	PC _S SBUS NONE, PmADR, IFCH, +2PC, IF AGLOW JUMP PWFAIL else STEP	:ADR← PC (adresa de fetch), IR← MEM ADR (fetch instrucțiune), PC← PC+2, test ACLOW (punct de ramificare în două direcții)
	01H	IF NCIL JUMP1 A+(CL ₁ , CL _D) else STEP	:test CL1 (cod legal), punct de ramificare în 2 direcții
ILLEGAL:	02H	A(0)BE, JUMP INT	:BE _I ← I (pentru generarea vectorului de excepție aferent), salt la microrutina de întrerupere (salt la eticheta INT)
PWFAIL:	03H	A(0)BE _I , JUMP INT	:BE _I ← I (pentru generarea vectorului de excepție aferent), salt la microrutina de întrerupere (salt la eticheta INT)
INT:	04H	SP _S SBUS NONE, PmADR,(INTA, -2SP), STEP	:ADR← SP← SP-2, INTA=I (achizițiere întrerupere; prim achizițare se va reseta și bistabilul de excepție sau de întrerupere setat)
	05H	FLAGS SBUS NONE, PmMDR, WRITE, -2SP, STEP	:MEM ADR← -FLAG (stiva← FLAG), SP← SP-2
	06H	SP _S SBUS NONE, PmADR, READ, STEP	:ADR← SP
	07H	PC _S SBUS NONE, PmMDR, WRITE, STEP	:MEM ADR← -PC (stiva← PC)
	08H	IVRS SBUS NONE, PmADR, READ, STEP	:ADR← IVR (ADR← vector de întrerupere / excepție).
	09H	MDR _S SBUS NONE, PmPC, (A(D)BE, A(D)BI), JUMP IF	:MDR← MEM ADR (MDR← adresa start <i>handler</i>)
A:	0AH	JUMPI FOSAD+(IR ₁ , IR _D)	:PC← MDR (PC← adresa start <i>handler</i>), BE← 0, BI← 0, salt la microrutina Instruction Fetch (salt la eticheta IF)
B:	0BH	JUMPI FODADB+(IR ₅ , IR _D)	:punct de ramificare pe setul B de microrutine Fetch/operand
C:	0CH	JUMPI BEQ+(IR ₁ +IR ₈ , 0)	: destinatie (microrutinile FOD aferente clasei B)
D:	0DH	JUMPI CIC+(IR ₁ +IR ₈ , 0)	:punct de ramificare pe microrutinile de execuție aferente clasei C
FOSAD:	0EH	RB _S SBUS NONE, PmT, JUMPI FODADA+(IR ₅ , IR _D)	:punct de ramificare pe microrutinile de execuție aferente clasei D ← operand sursa, punct de ramificare pe setul A de microrutine
			: Fetch operand destinatie (microrutinile FOD aferente clasei A)

F0SAM:	0F _H	PC _S SBUS NONE, PmADR, READ, +2PC, JUMP F0SEND	:ADR← PC, MDR← MEM ADR (MDR← operand sursa), PC← PC+2
F0SAL:	10 _H	RG _S SBUS NONE, PmADR, READ, JUMP F0SEND	:ADR← RG _S (ADR← adresa operand sursa), :MDR← MEM ADR (MDR← operand sursa)
F0SAX:	11 _H	PC _S SBUS NONE, PmADR, READ, +2PC, STEP	:ADR← PC, MDR← MEM ADR (MDR← index sursa), PC← PC+2
	12 _H	RG _S SUM MDR _D , PmADR, READ, STEP	:ADR← RG _S +MDR (ADR← registru sursa+index sursa), :MDR← MEM ADR (MDR← operand sursa)
F0SEND:	13 _H	MDR _S SBUS NONE, PmT, JUMP1 F0DADA+(IR ₅ , IR ₄)	:T← operand sursa, punct de ramificatie pe setul A de microrutine ;Fetch operand destinatie, punct de ramificatie pe microrutinele de executie aferente clasei A
F0DADA:	14 _H	NONE DBUS RG _D , PmMDR, JUMP1 MDV+(IR ₄ ÷IR ₂)	:MDR← operand destinatie, punct de ramificatie pe microrutinele de executie aferente clasei A
F0DAMA:	15 _H	JUMP ILLEGAL	;adresare imediata (AM) la destinatie este CIL (cod ilegal)
F0DAA:	16 _H	NONE DBUS RG _D , PmADR, READ, JUMP1 MDV+(IR ₄ ÷IR ₂)	:ADR← adresa operand destinatie, MDR← operand destinatie, punct ;de ramificatie pe microrutinile de executie aferente clasei A
F0DAXA:	17 _H	NONE DBUS PC _D , PmADR, READ, +2PC, STEP	:ADR← PC, MDR← MEM ADR (MDR← index destinatie), PC← PC+2
	18 _H	MDR _S SUM RG _D , PmADR, READ, JUMP1 MDV+(IR ₄ ÷IR ₂)	:ADR← MDR+RG _D (ADR← index destinatie+registru destinatie), :MDR← MEM ADR (MDR← operand destinatie), punct de ramificatie ;pe microrutinile de executie aferente clasei A
F0DADB:	19 _H	NONE DBUS RG _D , PmMDR, JUMP1 CLR+(IR ₁ ÷IR ₈)	:MDR← operand destinatie, punct de ramificatie pe microrutinile de executie aferente clasei B
F0DAMB:	1A _H	JUMP ILLEGAL	;adresare imediata (AM) la destinatie este CIL (cod ilegal)
F0DAB:	1B _H	NONE DBUS RG _D , PmADR, READ, JUMP1 MDV+(IR ₄ ÷IR ₂)	:ADR← adresa operand destinatie, MDR← operand destinatie, punct ;de ramificatie pe microrutinile de executie aferente clasei B
F0DAXB:	1C _H	NONE DBUS PC _D , PmADR, READ, +2PC, STEP	:ADR← PC, MDR← MEM ADR (MDR← index destinatie), PC← PC+2
	1D _H	MDR _S SUM RG _D , PmADR, READ, JUMP1 CLR+(IR ₁ ÷IR ₈)	:ADR← MDR+RG _D (ADR← index destinatie+registru destinatie), :MDR← MEM ADR (MDR← operand destinatie), punct de ramificatie ;pe microrutinile de executie aferente clasei B

ORGANIZAREA ŞI PROIECTAREA CALCULATOARELOR

MOV:	$1E_H$	$T_S \text{ SBUS } \text{NONE}, P_{mMDR}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow T$ ($MDR \leftarrow$ operand sursă), salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
ADD:	$1F_H$	$T_S \text{ SUM } MDR_0, P_{mMDR}, P_{dCOND_A}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow MDR+T$, pozitionare $\#g\text{-uri } C/Z/S/V$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
SUB:	20_H	$\bar{T}_S \text{ SUM } MDR_0, P_{mMDR}, (G_N, P_{dCOND_A}), \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow MDR-T$, pozitionare $\#g\text{-uri } C/Z/S/V$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
CMP:	21_H	$\bar{T}_S \text{ SUM } MDR_0, (G_N, P_{dCOND_A}), \text{JMP}1 \text{ IFC+}(INTR, 0, 0)$: $MDR \leftarrow T$ dacă pentru pozitionare $\#g\text{-uri } C/Z/S/V$, punct interruptibil OR: $T_S \text{ OR } MDR_0, P_{mMDR}, P_{dCOND}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$
AND:	22_H	$T_S \text{ AND } MDR_0, P_{mMDR}, P_{dCOND_L}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow MDR \text{ AND } T$, pozitionare $\#g\text{-uri } Z/S$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
XOR:	23_H	$T_S \text{ XOR } MDR_0, P_{mMDR}, P_{dCOND_L}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow MDR \text{ XOR } T$, pozitionare $\#g\text{-uri } Z/S$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
CLR:	24_H	$T_S \text{ NONE } DBUS_0, P_{mMDR}, P_{dCOND}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow 0$, pozitionare $\#g\text{-uri } Z/S$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
NEG:	25_H	$T_S \text{ NONE } DBUS_0, P_{mMDR}, P_{dCOND}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow \overline{MDR}$, pozitionare $\#g\text{-uri } Z/S$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
INC:	26_H	$T_S \text{ SUM } MDR_0, P_{mMDR}, P_{dCOND}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow MDR+1$, pozitionare $\#g\text{-uri } C/Z/S/V$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
DEC:	27_H	$T_S \text{ SUM } MDR_0, P_{mMDR}, P_{dCOND_A}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow MDR-1$, pozitionare $\#g\text{-uri } C/Z/S/V$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
ASL:	28_H	$-T_S \text{ SUM } MDR_0, P_{mMDR}, P_{dCOND_A}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow ASL(MDR)$, pozitionare $\#g\text{-uri } C/Z/S/V$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
ASR:	29_H	$T_S \text{ NONE } ASL MDR_0, P_{mMDR}, P_{dCOND_A}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow ASR(MDR)$, pozitionare $\#g\text{-uri } C/Z/S/V$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
LSR:	$2A_H$	$T_S \text{ NONE } ASR MDR_0, P_{mMDR}, P_{dCOND_A}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow LSR(MDR)$, pozitionare $\#g\text{-uri } Z/S$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
ROL:	$2B_H$	$T_S \text{ NONE } LSR MDR_0, P_{mMDR}, P_{dCOND_L}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow ROL(MDR)$, pozitionare $\#g\text{-uri } Z/S$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)
ROD:	$2C_H$	$T_S \text{ NONE } ROL MDR_0, P_{mMDR}, P_{dCOND_L}, \text{JMP}1 \text{ WRD+}(IR_5, IR_4)$: $MDR \leftarrow ROD(MDR)$, pozitionare $\#g\text{-uri } Z/S$, salt indexat la WRD (salt indexat la microurină de scriere rezultat la destinație)

CAPITOLUL 4 – Microprogramare. Proiectare avansată

ROR:	20H	NONE RDR MDR0, PmMDR, PdCOND, JUMP1 WRD+(IR5, IR4)	:MDR \leftarrow RD(RMDR), pozitionare flaguri Z/S, salt indexat la WRD (salt indexat la microrutina de scriere rezultat la destinatie)
RLC:	2EH	NONE RLC MDR0, PmMDR, PdCOND, JUMP1 WRD+(IR5, IR4)	:MDR \leftarrow RL(RMDR), pozitionare flaguri Z/S, salt indexat la WRD (salt indexat la microrutina de scriere rezultat la destinatie)
RRC:	2FH	NONE RRC MDR0, PmMDR, PdCOND, JUMP1 WRD+(IR5, IR4)	:MDR \leftarrow RR(RMDR), pozitionare flaguri Z/S, salt indexat la WRD (salt indexat la microrutina de scriere rezultat la destinatie)
PUSH:	30H	SP \downarrow SBUS NONE, PmADR, WRITE, -2SP, JUMP1 IFC+(INTR, 0, 0)	:ADR \leftarrow SP-2 (-2SP de tip nivel pt. a încărca în ADR valoarea SP-2), :MEM ADR \leftarrow MDR (scriere în stiva)
POP:	31H	ADRS SBUS NONE, PmT, STEP	:salvare ADR în T pt. a face POP functional cu toate modurile de adresare
	32H	SP \downarrow SBUS NONE, PmADR, READ, STEP	:ADR \leftarrow SP, MDR \leftarrow MEM ADR (citire locatie din vârful stivei)
	33H	T _S , SBUS NONE, PmADR, +2SP, JUMP1 WRD+(IR5, IR4)	:ADR \leftarrow T (restaurare ADR din T), SP \leftarrow SP+2 (noul vârf al stivei), :salt indexat la WRD (POP functional cu toate modurile de adresare)
WRD:	34H	MDR _S SBUS NONE, PmRG, JUMP1 IFC+(INTR, 0, 0)	:RG \leftarrow MDR (Registru destinatie \leftarrow Rezultat) -adresare directă (AD) :destinatie cu adresare imediata (AM) este CL (cod ilegal)
	35H	JUMP ILLEGAL	:MEM ADR \leftarrow MDR (Memorie \leftarrow Rezultat) -adresare indirectă (A)
	36H	WRITE, JUMP1 IFC+(INTR, 0, 0)	:punct interuptibil
	37H	WRITE, JUMP1 IFC+(INTR, 0, 0)	:MEM ADR \leftarrow MDR (Memorie \leftarrow Rezultat) -adresare indexata (AX) :punct interuptibil
BEQ:	38H	IF Z JUMP1 JMPAD+(IR5, IR4) else STEP	:dacă Z=1: salt la executie JMP cu AM (salt direct), JMP cu AD (CL), .JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	39H	JUMP1 IFC+(INTR, 0, 0)	:dacă Z=0: punct interuptibil (conditie de salt neîndeplinita)
BNE:	3AH	IF NZ JUMP1 JMPAD+(IR5, IR4) else STEP	:dacă Z=0: salt la executie JMP cu AM (salt direct), JMP cu AD (CL), .JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	3BH	JUMP1 IFC+(INTR, 0, 0)	:dacă Z=1: punct interuptibil (conditie de salt neîndeplinita)
BM:	3CH	IF S JUMP1 JMPAD+(IR5, IR4) else STEP	:dacă S=1: salt la executie JMP cu AM (salt direct), JMP cu AD (CL), .JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	3DH	JUMP1 IFC+(INTR, 0, 0)	:dacă S=0: punct interuptibil (conditie de salt neîndeplinita)

ORGANIZAREA ŞI PROIECTAREA CALCULATOARELOR

BPL:	3F _H	IF NS JUMPI JMPAD+(IR ₅ , IR ₄) else STEP	:daca S=0: sait la executie JMP cu AM (salt direct). JMP cu AD (CIL). :JMP cu AI (salt indirect). JMP cu AX (salt indexat)
	3F _H	JUMPI IF<-(INTR, 0, 0)	:daca S=1: punct interuptibil (conditie de salt neindexata)
BCS:	40 _H	IF C JUMPI JMPAD+(IR ₅ , IR ₄) else STEP	:daca C=1: sait la executie JMP cu AM (salt direct). JMP cu AD (CIL). :JMP cu AI (salt indirect). JMP cu AX (salt indexat)
	41 _H	JUMPI IF>-(INTR, 0, 0)	:daca C=0: punct interuptibil (conditie de salt neindexata)
BCC:	42 _H	IF NC JUMPI JMPAD+(IR ₅ , IR ₄) else STEP	:daca C=0: sait la executie JMP cu AM (salt direct). JMP cu AD (CIL). :JMP cu AI (salt indirect). JMP cu AX (salt indexat)
	43 _H	JUMPI IF<-(INTR, 0, 0)	:daca C=1: punct interuptibil (conditie de salt neindexata)
BVS:	44 _H	IF V JUMPI JMPAD+(IR ₅ , IR ₄) else STEP	:daca V=1: sait la executie JMP cu AM (salt direct). JMP cu AD (CIL). :JMP cu AI (salt indirect). JMP cu AX (salt indexat)
	45 _H	JUMPI IF>-(INTR, 0, 0)	:daca V=0: punct interuptibil (conditie de salt neindexata)
BVC:	46 _H	IF NV JUMPI JMPAD+(IR ₅ , IR ₄) else STEP	:daca V=0: sait la executie JMP cu AM (salt direct). JMP cu AD (CIL). :JMP cu AI (salt indirect). JMP cu AX (salt indexat)
	47 _H	JUMPI IF<-(INTR, 0, 0)	:daca V=1: punct interuptibil (conditie de salt neindexata)
JMP:	48 _H	JUMPI JMPAD+(IR ₅ , IR ₄)	:sait la executie JMP cu AM (salt direct). JMP cu AD (CIL). :JMP cu AI (salt indirect). JMP cu AX (salt indexat)
	49 _H	NONE	
			:salt indexat la executie: CALL cu AD (CIL), CALL cu AM (call direct).
CALL:	4A _H	JUMPI CALLAD+(IR ₅ , IR ₄)	:CALL cu AI (call indirect). CALL cu AX (call indexat)
CALLAD:	4B _H	JUMP ILLEGAL	:CALL cu adresaare directa este cod ilegal (CIL)
CALLAM:	4C _H	PC _S SBUS NONE, PmADR, READ, +2PC, JUMP CALL2	:ADR _C → PC, MDR→ MEM ADR (MDR→ Offset), PC→ PC+2
CALLA1:	4D _H	NONE DBUS Rg _D , PmT, JUMP CALL3	:T→ Rg _D (T→ adresa de call)
CALLAX:	4E _H	PC _S SBUS NONE, PmADR, READ, +2PC, STEP	:ADR _C → PC, MDR→ MEM ADR (MDR→ Index), PC→ PC+2
	4F _H	MDRS SUM Rg _D , PmT, JUMP CALL3	:T→ MDRS+Rg _D (T→ adresa de call; adresa de call=Rg _D +index)
CALLZ:	50 _H	MDRS SUM PC _D , PmT, STEP	:T→ MDRS+PC (T→ adresa de call; adresa de call=PC+Offset)

CAPITOLUL 4 – Microprogramare. Proiectare avansată

CALLB:	51H	SP ₃ SBUS NONE, PmADR, -2SP, STEP	:ADR← SP-2, (-2SP comanda de tip nivel astfel încât în ADR să se incarce valoarea obținuta în SP după decrementare)
	52H	PC ₃ SBUS NONE, PmADR, WRITE, STEP	:MEM ADR← PC (Stiva← PC, PC← adresa salvată în stivă)
	53H	T ₃ SBUS NONE, PmPC, JUMP IFC+(INTR, 0, 0)	:PC← T (PC← adresa de call), punct interuptibil
JMPAD:	54H	JUMP ILLEGAL	:JMP cu adresare directă este cod ilegal (CIL)
JMPAM:	55H	PC ₃ SBUS NONE, PmADR, READ, +2PC, JUMPAM2	:ADR← PC, MDR← MEM ADR (MDR← Offset), PC← PC+2
JMPAI:	56H	NONE DBUS R ₅₉ , PmPC, JUMP IFC+(INTR, 0, 0)	:PC← R ₅₉ (PC← adresa de salt), punct interuptibil
JMPAX:	57H	PC ₃ SBUS NONE, PmADR, READ, +2PC, STEP	:ADR← PC, MDR← MEM ADR (MDR← Index), PC← PC+2
	58H	MDR ₃ SUM R ₆₀ , PmPC, JUMP IFC+(INTR, 0, 0)	:PC← MDR+R ₆₀ (PC← adresa de salt; adresa de salt=R ₆₀ +index)
JMPAMZ:	59H	MDR ₃ SUM PC ₀ , PmPC, JUMP IFC+(INTR, 0, 0)	:PC← MDR+PC (PC← adresa de salt; adresa de salt=PC+Offset)
CLC:	5AH	FLAGS AND IR7÷0, PmFLAG, JUMP IFC+(INTR, 0, 0)	:FLAG← FLAG AND IR7÷0 (IR7÷0=Masca), punct interuptibil (bitii pe 0 în masca desemnează flagurile ce vor fi resetate)
	5BH	NONE	
SET:	5CH	FLAGS OR IR7÷0, PmFLAG, JUMP IFC+(INTR, 0, 0)	:FLAG← FLAG OR IR7÷0 (IR7÷0=Masca), punct interuptibil (bitii pe 1 în masca desemnează flagurile ce vor fi setate)
	5DH	NONE	
NDP:	5EH	JUMP IFC+(INTR, 0, 0)	:punct interuptibil
	5FH	NONE	
HALT:	60H	A(D)RPO	:RPO← 0 (resetare bistabil porneire-ședere pt. blocare tact procesor)
	61H	NONE	
EI:	62H	A(I)BVI, JUMP IFC+(INTR, 0, 0)	:BVI← 1 (setare bistabil de validare a întreverperilor), punct interupt.
	63H	NONE	
DI:	64H	A(D)BVI, JUMP IFC	:BVI← 0 (resetare bistabil de validare a întreverperilor), salt la IF
PUSHPC:	65H	SP ₃ SBUS NONE, PmADR, -2SP, STEP	:ADR← SP-2 (-2SP de tip nivel pt. a încărca în ADR valoarea SP-2)
	66H		

ORGANIZAREA ŞI PROIECTAREA CALCULATOARELOR

	67H	PC ₃ SBUS NONE, P _m MDR, WRITE, JUMPI FC+(INTR, 0, 0)	:MEM ADR←-PC; (Stiva←-PC), punct interruptibil
POP PC;	68H	SP ₃ SBUS NONE, P _m ADR, READ, STEP	:ADR←-SP, MDR←-MEM ADR (MDR←-continut locatie din varful stivei)
	69H	MDR ₃ SBUS NONE, P _m PC, +2SP, JUMPI FC+(INTR, 0, 0)	:PC←-MDR, SP←-SP-2, punct interruptibil
PUSH FLAG;	6AH	SP ₃ SBUS NONE, P _m ADR, -2SP, STEP	:ADR←-SP-2 (-2SP de tip nivel pt. a incarcă în ADR valoarea SP-2)
	6BH	FLAGS SBUS NONE, P _m MDR, WRITE, JUMPI FC+(INTR, 0, 0)	:MEM ADR←-FLAG (Stiva←-FLAG), punct interruptibil
POP FLAG;	6CH	SP ₃ SBUS NONE, P _m ADR, READ, STEP	:ADR←-SP, MDR←-MEM ADR (MDR←-continut locatie din varful stivei)
	6DH	MDR ₃ SBUS NONE, P _m FLAG, +2SP, JUMPI FC+(INTR, 0, 0)	:FLAG←-MDR, SP←-SP-2, punct interruptibil
RET;	6EH	SP ₃ SBUS NONE, P _m ADR, READ, STEP	:ADR←-SP, MDR←-MEM ADR (MDR←-continut locatie din varful stivei)
	6FH	MDR ₃ SBUS NONE, P _m PC, +2SP, JUMPI FC+(INTR, 0, 0)	:PC←-MDR, SP←-SP-2, punct interruptibil
IRET;	70H	SP ₃ SBUS NONE, P _m ADR, READ, STEP	:ADR←-SP, MDR←-MEM ADR (MDR←-continut locatie din varful stivei)
	71H	MDR ₃ SBUS NONE, P _m PC, +2SP, STEP	:PC←-MDR, SP←-SP-2
	72H	SP ₃ SBUS NONE, P _m ADR, READ, STEP	:ADR←-SP, MDR←-MEM ADR (MDR←-continut locatie din varful stivei)
	73H	MDR ₃ SBUS NONE, P _m FLAG, +2SP, JUMPI FC+(INTR, 0, 0)	:FLAG←-MDR, SP←-SP-2, punct interruptibil

Tabelul 4.9 Microprogramul de emulare aferent procesorului CISCC

Microprogramul de emulare (tabelul 4.9) a fost elaborat în concordanță cu schema de organizare din figura 4.17. Microinstrucțiunile sunt specificate în limbajul mnemonic adoptat și pentru fiecare microinstrucțiune, pe coloana de comentarii, sunt explicitate succint operațiile efectuate de respectiva microinstrucțiune.

Emularea unei instrucțiuni presupune parcurgerea unei succesiuni de microrutine. Succesiunea începe întotdeauna cu microrutina IF (*Instruction Fetch*) dar continuarea (seria de microrutine care urmează) depinde de clasa din care face parte instrucțiunea. După cum rezultă și din figura 4.17, pentru emularea unei instrucțiuni din clasa A se va parcurge o succesiune de 4 microrutine, pentru emularea unei instrucțiuni din clasa B se va parcurge o succesiune de 3 microrutine iar pentru emularea unei instrucțiuni din clasele C și respectiv D, se va parcurge o succesiune de 2 microrutine.

Cele 4 microrutine care se succed în timpul emulării unei instrucțiuni din clasa A, precum și legăturile dintre aceste microrutine, implementate prin salturi indexate, sunt explicitate în figura 4.32. Schema explicativă sintetizată în această figură trebuie analizată în paralel cu microprogramul de emulare din tabelul 4.19. Prin urmare, cele 4 microrutine parcuse pentru emularea unei instrucțiuni din clasa A (a unei instrucțiuni cu 2 operanzi) sunt:

1. **Microrutina IF** (*Instruction Fetch*) care va extrage (va citi) instrucțiunea din memorie și o va încărca în procesor (în registrul IR). În microprogramul de emulare (tabelul 4.9), microrutina **IF** începe la microadresa 00_H (eticheta IFC). Imediat după încărcarea instrucțiunii în registrul IR, aceasta va fi decodificată, astfel încât pe ieșirile decodificatorului instrucțiunii se vor activa semnalele care descriu tipul instrucțiunii. Printre aceste semnale, vom regăsi și semnalele CL_1 , CL_0 , care codifică clasa din care face parte instrucțiunea. Este evident că următoarea microinstrucțiune (microinstrucțiunea de la microadresa 01_H) beneficiază de semnale CL_1 , CL_0 stabile și corecte și prin urmare va executa corect saltul indexat JUMPI A+(CL_1 , CL_0). Acest salt indexat implementează ramificația pe cele 4 direcții majore (A, B, C, D), direcții asignate celor 4 clase de instrucțiuni.
2. **Microrutina FOS** (*Fetch Operand Sursă*) care va localiza operandul sursă, îl va citi (din memorie dacă este cazul) și-l va încărca în procesor (în registrul T). Microrutina **FOS** este parcursă numai dacă instrucțiunea emulată este din clasa A. În realitate, există 4 microrutine **FOS** (câte una pentru fiecare mod de adresare):
 - microrutina **FOS** pentru adresare directă, care începe la microadresa $0E_H$ (eticheta FOSAD)
 - microrutina **FOS** pentru adresare imediată, care începe la microadresa $0F_H$ (eticheta FOSAM)
 - microrutina **FOS** pentru adresare indirectă, care începe la microadresa 10_H (eticheta FOSAI)
 - microrutina **FOS** pentru adresare indexată, care începe la microadresa 11_H (eticheta FOSAX)

ORGANIZAREA ȘI PROIECTAREA CALCULATOARELOR

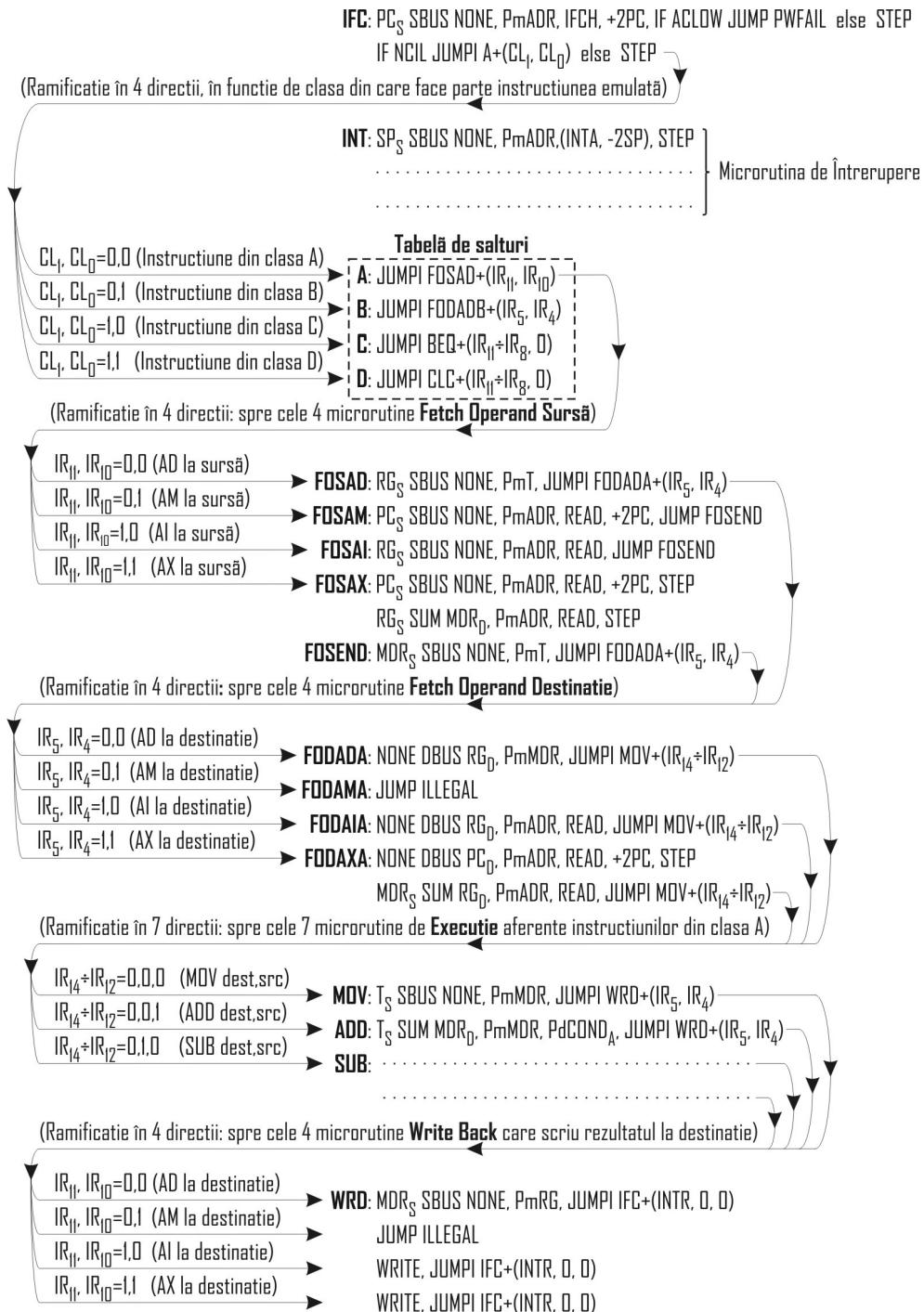


Fig 4.32 Mecanismul de selectie a microrutinelor în cadrul procesului de emulare a unei instructiuni din clasa A

Pentru localizarea operandului sursă al instrucțiunii, cele 4 microrutine **FOS** operează în acord cu schemele de localizare prezentate în figurile 4.8, 4.9, 4.10 și 4.11. Legătura între microrutina **IF** și cele 4 microrutine **FOS** este implementată prin două salturi indexate successive. Primul salt este **JUMPI A+(CL₁, CL₀)** și se află la microadresa **01_H** (tabelul 4.9). Acest succesor realizează saltul într-o tabelă de salturi cu dimensiunea de 4 locații (4 microinstructiuni). De fapt succesorul **JUMPI A+(CL₁, CL₀)** realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa A, atunci **CL₁, CL₀=0,0** și succesorul **JUMPI A+(CL₁, CL₀)** va realiza saltul în prima locație a tabelei de salturi (la microadresa **0A_H**). Aici regăsim cel de-al doilea salt indexat, **JUMPI FOSAD+(IR₁₁, IR₁₀)**, care implementează legătura spre cele 4 microrutine **FOS** (vezi fig 4.32).

- 3. Microrutina **FOD_A**** (*Fetch Operand Destinație*) care va localiza operandul destinație, îl va citi (din memorie dacă este cazul) și-l va încărca în procesor (în registrul MDR). Există 3 microrutine **FOD_A** (câte una pentru fiecare mod de adresare acceptat la destinație):

- microrutina **FOD_A** pentru adresare directă, care începe la microadresa **14_H** (eticheta **FODADA**)
- microrutina **FOD_A** pentru adresare indirectă, care începe la microadresa **16_H** (eticheta **FODAIA**)
- microrutina **FOD_A** pentru adresare indexată, care începe la microadresa **17_H** (eticheta **FODAXA**)

Pentru localizarea operandului destinație al instrucțiunii, cele 3 microrutine **FOD_A** operează în acord cu schemele de localizare prezentate în figurile 4.8, 4.10 și 4.11.

Reamintim faptul că, adresarea imediată la destinație reprezintă un nonsens și de aceea este considerată ilegală (vezi 4.1.3.B). Din acest motiv nu există o microrutină **FOD_A** cu adresare imediată și prin urmare, la eticheta **FODAMA** (microadresa **15_H**) regăsim o microinstructiune de salt (**JUMP ILLEGAL**) care face trimitere la tratarea excepției **CIL** (instrucțiune ilegală!).

Legătura dintre microrutinile **FOS** și microrutinile **FOD_A** este realizată prin saltul indexat **JUMPI FODADA+(IR₅, IR₄)** pe care-l regăsim în microprogram la microadresele **0E_H** și **13_H** (vezi figura 4.32 și tabelul 4.9).

- 4. Microrutina **EX_A**** (de Execuție) în care se efectuează operația (revendicată de instrucțiune) asupra celor 2 operanzi, se generează rezultatul și se memorează acest rezultat la destinație (se suprascrie operandul destinație). Există 7 instrucțiuni în clasa A: (MOV, ADD, SUB, CMP, AND, OR și XOR). Prin urmare, în microprogram pot fi identificate 7 microrutine **EX_A** (câte una pentru fiecare instrucțiune). Cele 7 microrutine încep la etichete identice cu mnemonicele

instrucțiunilor: MOV (microadresa $1E_H$), ADD (microadresa $1F_H$), SUB (microadresa 20_H), CMP (microadresa 21_H), AND (microadresa 22_H), OR (microadresa 23_H) și respectiv XOR (microadresa 24_H). Microinstructiunile de la cele 7 etichete precizate, efectuează operația aritmetică-logică (revendicată de instrucțiune), poziționează *flag*-urile de condiții în acord cu rezultatul obținut (excepție face instrucțiunea MOV care nu poziționează *flag*-urile) și execută saltul indexat JUMPI WRD+(IR₅, IR₄). Acest salt indexat reprezintă un punct de ramificație în 4 direcții. Prin urmare, în funcție de valoarea câmpului MAD (valoarea câmpului modul de adresare destinație codificat în instrucțiunea emulată), saltul se va face la următoarele microadrese:

- la microadresa 34_H (eticheta WRD), dacă în câmpul MAD este codificată adresare directă ($IR_{5,4}=0,0$)
- la microadresa 35_H , dacă în câmpul MAD este codificată adresare imediată ($IR_{5,4}=0,1$)
- la microadresa 36_H , dacă în câmpul MAD este codificată adresare indirectă ($IR_{5,4}=1,0$)
- la microadresa 37_H , dacă în câmpul MAD este codificată adresare indexată ($IR_{5,4}=1,1$)

Menționăm că adresarea imediată la destinație este din nou tratată ca și instrucțiune ilegală. Din acest motiv, la microadresa 35_H , regăsim saltul necondiționat: JUMP ILLEGAL.

La celelalte 3 microadrese (34_H , 36_H și 37_H) se realizează scrierea rezultatului la destinație. Din acest motiv, în figura 4.32 le-am denumit microroutine *write back*. Aceste microroutine *write back* conțin câte o singură microinstructiune și reprezintă de fapt partea finală a microroutinei **EX_A**. Pe cele 3 microinstructiuni care realizează operația *write back* este implementat și punctul interruptibil. Asta e, s-a finalizat execuția instrucțiunii și la sfârșitul fiecărei instrucțiuni trebuie implementat punctul interruptibil!

Relativ la operația de scriere a rezultatului la destinație, trebuie să subliniem faptul că în cazul AD, rezultatul se scrie în registrul destinație (microcomanda PmRG de la microadresa 34_H) iar în cazurile AL și AX, rezultatul se scrie în memorie, la adresa existentă în registrul ADR (microcomanda WRITE de la microadresele 36_H și respectiv 37_H). Cum în microrutina anterioară (în microrutina **FOD_A**), în registrul ADR a fost încărcată adresa operandului destinație, este evident că rezultatul produs de instrucțiune va fi scris la destinație (se suprascrie operandul destinație).

Să observăm că punctul interruptibil de la sfârșitul instrucțiunii (vezi microadresele 34_H , 36_H și 37_H) este implementat tot printr-un salt indexat:

JUMPI IFC+(INTR, 0, 0)

Acest salt indexat reprezintă un punct de ramificație în 2 direcții. Prin urmare, saltul se va face:

- la microadresa 00_H (la eticheta IFC), dacă $\text{INTR}=0$. Prin urmare, dacă nu există nici-o întrerupere activă, se trece la *fetch*-ul următoarei instrucțiuni din cadrul programului aflat în execuție.
- la microadresa 04_H (eticheta INT), dacă $\text{INTR}=1$. Prin urmare, dacă cel puțin un periferic a activat cererea de întrerupere și întreruperile sunt validate, se trece la microrutina de întrerupere. Aceasta va suspenda temporar *task*-ul (programul) curent și va lansa în execuție *handler*-ul de întrerupere aferent perifericului care a solicitat întreruperea. *Handler*-ul va face perifericului solicitant, serviciul solicitat. Evident că și instrucțiunile *handler*-ului, ca de altfel ale oricărui program, sunt emulate în microcod (procesate sub controlul microprogramului de emulare).

Să precizăm că instrucțiunea **CMP** compară (prin scădere) cei 2 operanzi, sursă și respectiv destinație, poziționează *flag*-urile de condiții în acord cu rezultatul comparației (scăderii), dar nu depune rezultatul la destinație! În fond, prin aceasta se deosebește de instrucțiunea **SUB** (vezi microadresa 21_H din tabelul 4.19).

Legătura dintre microrutinile **FOD_A** și microrutinile **EX_A** este implementată prin saltul indexat **JUMPI MOV+(IR₁₄÷IR₁₂)** (vezi microadresele 14_H , 16_H și 18_H în tabelul 4.9 și respectiv figura 4.32). Indexul utilizat pentru implementarea acestei ramificații în 7 direcții (spre cele 7 microrutine de execuție aferente instrucțiunilor din clasa A) este tocmai câmpul **OPCODE** al instrucțiunilor din clasa A (fig. 4.14).

După cum am precizat deja, mecanismul de selecție a microrutinelor din figura 4.32 sintetizează procesul de emulare a instrucțiunilor din clasa A iar în tabelul 4.9 regăsim microrutinile de emulare în extenso.

Pentru emularea unei instrucțiuni din clasa B se parurge o succesiune de 3 microrutini. Microrutinile parcuse, precum și legăturile dintre aceste microrutini, implementate prin salturi indexate, sunt explicitate în figura 4.33.

Instrucțiunile din clasa B sunt instrucțiuni cu un singur operand. Acest operand unic are semnificația de operand destinație (vezi figura 4.5 cu explicațiile aferente). Prin urmare, emularea unei instrucțiuni din clasa B diferă de emularea unei instrucțiuni din clasa A, prin faptul că la clasa B nu se mai parurge microrutina **FOS** (*Fetch Operand Sursă*). Cele 3 microrutini parcuse în cadrul clasei B sunt:

- **IF** (microrutina *Instruction Fetch*)
- **FOD_B** (microrutina *Fetch Operand Destinație*)
- **EX_B** (microrutina de execuție aferentă instrucțiunii din clasa B). Ca și la instrucțiunile din clasa A, microrutina **EX_B** se încheie cu aceeași operație *write back*. În figurile 4.32 și 4.33 am utilizat chiar denumirea de **microrutină write back**. Privită sau nu ca o microrutină distinctă, reprezintă desigur ultima operație din cadrul microrutinei de Execuție, operația de depunere a rezultatului la destinație.

Similar cu microrutinile **FOD_A**, vom avea evident 3 microrutini **FOD_B** (câte una pentru fiecare mod de adresare viabil la destinație. Acestea se regăsesc în tabelul 4.19 la microadresele: 19_H (eticheta FODADB), $1B_H$ (eticheta FODAIB) și respectiv $1C_H$ (eticheta FODAXB). Menționăm încă o dată că adresarea imediată la destinație este

ORGANIZAREA ȘI PROIECTAREA CALCULATOARELOR

considerată ilegală (vezi 4.1.3.B). Din acest motiv nu există o microrutină **FOD_B** cu adresare imediată și prin urmare, la eticheta FODAMB (microadresa 1A_H) regăsim o microinstrucțiune de salt (JUMP ILLEGAL) care face trimitere la tratarea excepției CIL (instrucțiune ilegală!).

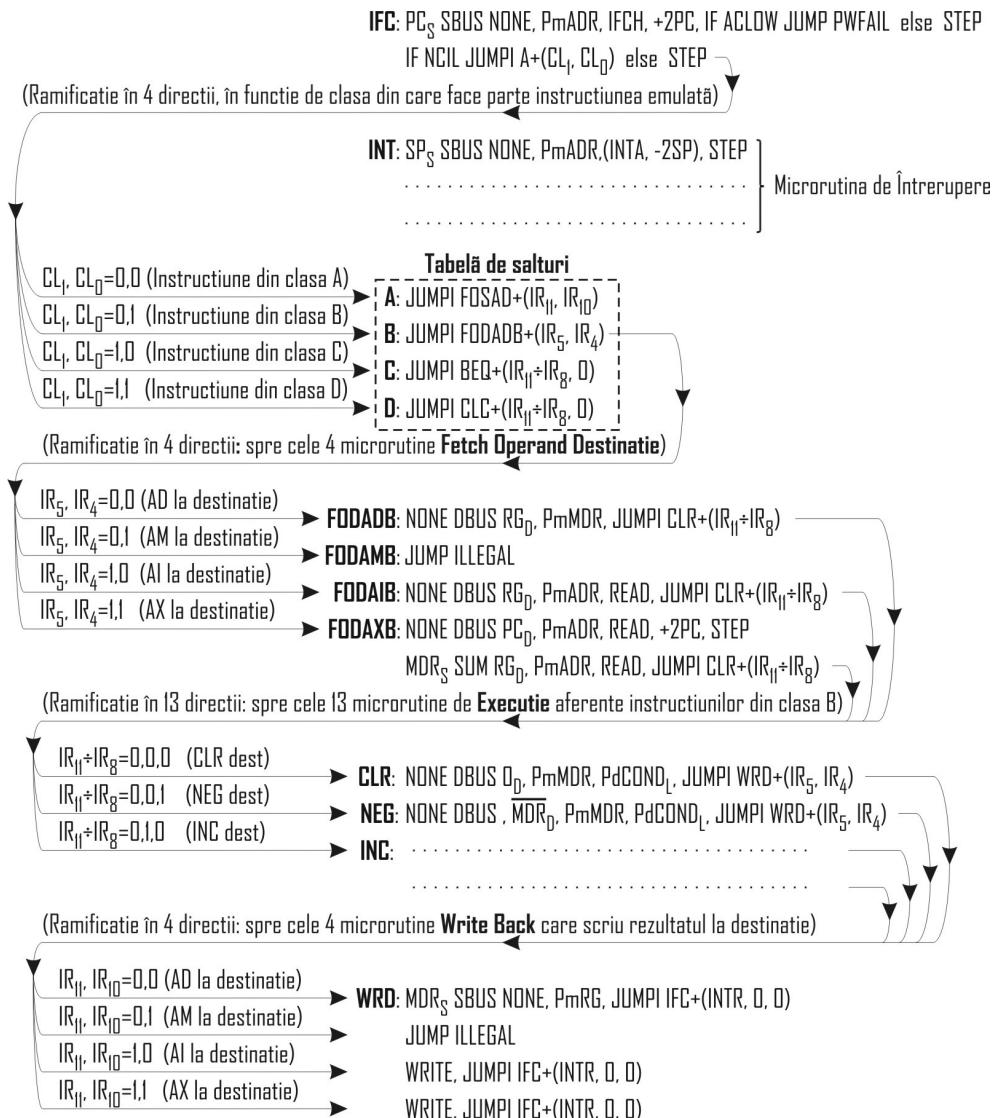


Fig 4.33 Mecanismul de selecție a microrutinelor în cadrul procesului de emulare a unei instrucțiuni din clasa B

Microrutinile **FOD_B** reprezintă copii aproape identice ale microrutinelor **FOD_A**. Singura diferență apare la nivelul legăturii cu microrutinile de execuție. Microrutinile **FOD_A** sunt urmate de microrutine **EX_A** (microrutine de execuție aferente instrucțiunilor din clasa A) iar microrutinile **FOD_B** sunt urmate de microrutine **EX_B** (microrutine de execuție aferente instrucțiunilor din clasa B). Prin urmare, legătura spre microrutinile de execuție a fost implementată cu saltul indexat:

JUMPI MOV+(IR₁₄÷IR₁₂), în cazul microrutinelor **FOD_A**.

JUMPI CLR+(IR₁₁÷IR₈), în cazul microrutinelor **FOD_B**.

În concluzie, dacă facem abstracție de diferență care apare la nivelul succesorilor utilizati, cele 4 microrutine **FOD_B** reprezintă copii identice ale microrutinelor **FOD_A**. Am decis implementarea a două seturi identice de microrutine **FOD** (**FOD_A** și **FOD_B**) pentru a simplifica intrarea în microrutinile **EX_A** și respectiv **EX_B**. Dacă am fi utilizat același set de microrutine **FOD** pentru ambele clase de instrucțiuni, atunci, la ieșirea din microrutinile **FOD**, ar fi trebuit implementate două mecanisme de ramificație successive: primul în două direcții (clasa A și respectiv B) și al doilea pe microrutina aferentă instrucțiunii din cadrul clasei. O astfel de ramificație prin două salturi indexate successive reprezintă evident o soluție mai neperformantă, comparativ cu soluția adoptată. Dezavantajul soluției adoptate (soluția cu două seturi de microrutine **FOD**) constă într-un consum suplimentar de memorie MPM care, pe baza microprogramului de emulare redat în tabelul 4.9, putem constata că este nesemnificativ.

Legătura între microrutina **IF** și cele 4 microrutine **FOD_B** (3 viabile și a patra reprezentând instrucțiune ilegală) este implementată prin două salturi indexate successive. Primul salt este JUMPI A+(CL₁, CL₀) și se află la microadresa 01_H (tabelul 4.9). Acest succesor realizează saltul într-o tabelă de salturi cu dimensiunea de 4 locații (4 microinstrucțiuni). De fapt succesorul JUMPI A+(CL₁, CL₀) realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa B, atunci CL₁, CL₀=0,1 și succesorul JUMPI A+(CL₁, CL₀) va realiza saltul în cea de a doua locație a tabelei de salturi (la microadresa 0B_H). Aici regăsim cel de-al doilea salt indexat, JUMPI FODADB+(IR₅, IR₄), care implementează legătura spre cele 4 microrutine **FOD_B** (vezi fig 4.33).

În clasa C regăsim instrucțiunile de salt:

- 8 instrucțiuni de salt condiționat (BEQ, BNE, BMI, BPL, BCS, BCC, BVS și BVC)
- o instrucțiune de salt necondiționat (JMP).
- o instrucțiune de apel procedură (CALL)

Pentru emularea unei instrucțiuni din clasa C se parcurge o succesiune de 2 microrutine:

- **IF** (microrutina *Instruction Fetch*)
- **EX_C** (microrutina de execuție aferentă instrucțiunii din clasa C).

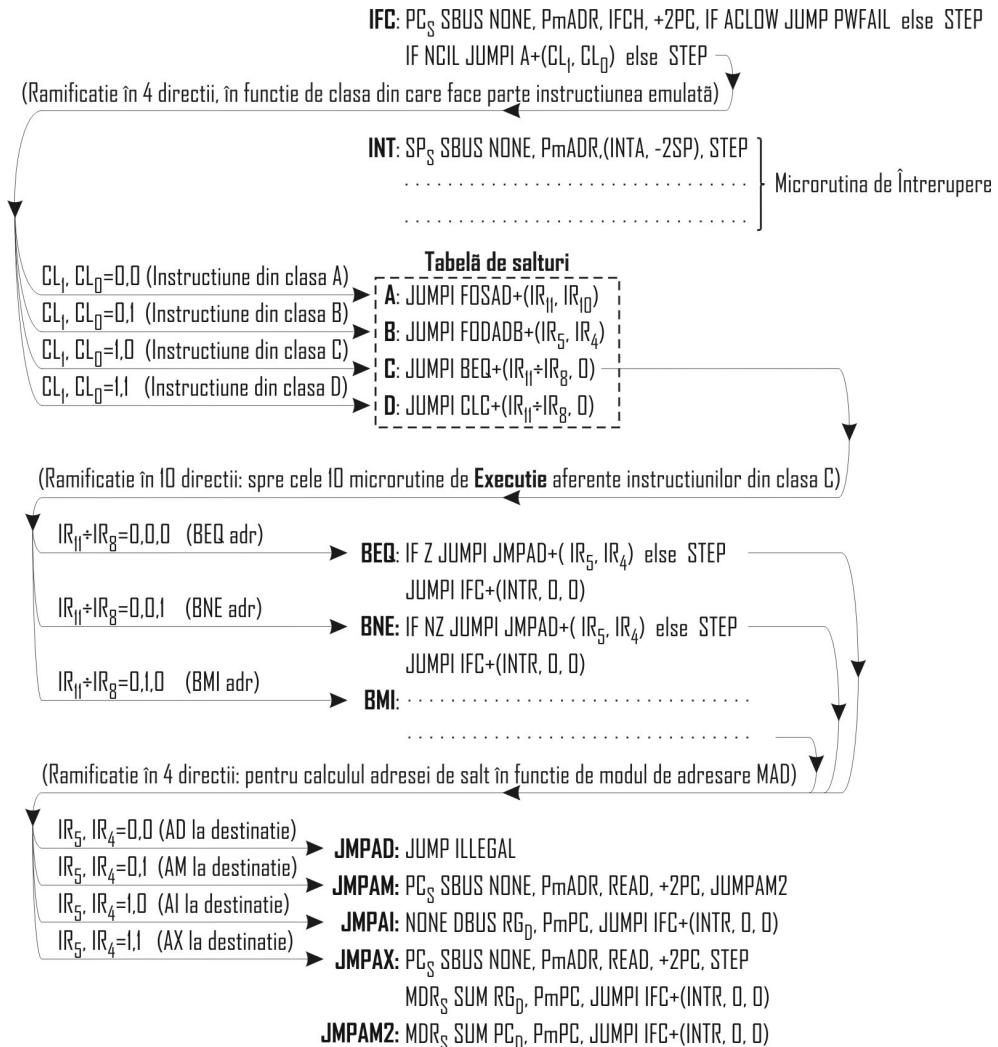


Fig 4.34 Mecanismul de selecție a microrutinelor în cadrul procesului de emulare a unei instrucțiuni de salt

Legătura între microrutina **IF** și cele 10 microrutine **ExC** este implementată prin două salturi indexate succesive. Primul salt este **JUMPI A+(CL₁, CL₀)** și se află la microadresa 01_H (tabelul 4.9). Acest succesor realizează saltul într-o tabelă de salturi cu dimensiunea de 4 locații (4 microinstructiuni). De fapt succesorul **JUMPI A+(CL₁, CL₀)** realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa C, atunci CL₁, CL₀=1,0 și succesorul **JUMPI A+(CL₁, CL₀)** va realiza saltul în cea de a treia locație a tabeliei de salturi (la microadresa

$0C_H$). Aici regăsim o altă microinstrucțiune de salt indexat, JUMPI BEQ+(IR₁₁÷IR₈, 0), care implementează legătura spre cele 10 microrutine **EX_C** (vezi fig 4.34).

Microrutinile **EX_C** emulează faza de execuție aferentă instrucțiunilor de salt (vezi 4.1.2.C). Reamintim că instrucțiunea CALL este în final tot o instrucțiune de salt. Aceasta realizează saltul pe prima instrucțiune a procedurii apelate. Evident că înainte de salt, ea va salva registrul PC (adresa de revenire) în stivă.

Prin urmare, în microrutinile **EX_C**, se calculează adresa de salt (respectiv de *call* la instrucțiunea CALL) și se va încărca această adresă în PC. La instrucțiunile de salt condiționat, încărcarea adresei de salt în PC este condiționată de îndeplinirea condiției testate; dacă condiția testată este neîndeplinită atunci, adresa de salt nu va fi încărcată în PC, saltul devenind inefectiv (se va continua programul pe ramura secvențială).

Calculul adresei de salt se face în funcție de modul de adresare codificat în instrucțiune (vezi fig 4.34 și tabelul 4.9). Acest calcul se face la etichetele: JMPAM (microadresa 55_H), JMPAI (microadresa 56_H), și respectiv JMPAX (microadresa 57_H). Modul de adresare direct este ilegal la instrucțiunile de salt. Acest lucru a fost justificat la 4.1.2.C și respectiv implementat în microcod la eticheta JMPAD (microadresa 54_H).

Calculul adresei de *call* (la instrucțiunea CALL) se face în funcție de modul de adresare codificat în instrucțiune (vezi fig 4.34 și tabelul 4.9). Acest calcul (practic identic cu cel prezentat deja la instrucțiunile de salt) se face la etichetele: CALLAM (microadresa $4C_H$), CALLAI (microadresa $4D_H$), și respectiv CALLAX (microadresa $4E_H$). Modul de adresare direct este evident ilegal și la instrucțiunea CALL. Acest lucru a fost justificat la 4.1.2.C și respectiv implementat în microcod la eticheta CALLAD (microadresa $4B_H$).

În clasa D regăsim instrucțiunile fără operanți (vezi 4.1.2.D). Într-o manieră similară celei prezentate în cazul anterior (clasa instrucțiunilor de salt), pentru emularea unei instrucțiuni din clasa D se parcurge o succesiune de 2 microrutine:

- **IF** (microrutina *Instruction Fetch*)
- **EX_D** (microrutina de execuție aferentă instrucțiunii din clasa D).

Legătura între microrutina **IF** și cele 20 (în realitate 12!) microrutine **EX_D** este implementată prin două salturi indexate succesive. Primul salt este JUMPI A+(CL₁, CL₀) și se află la microadresa 01_H (tabelul 4.9). Acest succesor realizează saltul într-o tabelă de salturi cu dimensiunea de 4 locații (4 microinstrucțiuni). De fapt succesorul JUMPI A+(CL₁, CL₀) realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa D, atunci CL₁, CL₀=1,1 și succesorul JUMPI A+(CL₁, CL₀) va realiza saltul în cea de a patra locație a tabelei de salturi (la microadresa $0D_H$). Aici regăsim o altă microinstrucțiune de salt indexat, JUMPI CLC+(IR₁₁÷IR₈, 0), care implementează legătura spre cele 20 (în realitate 12!) microrutine **EX_D** (vezi fig 4.35 și microprogramul de emulare din tabelul 4.9).

Deși avem 20 instrucțiuni în clasa D, în microprogramul de emulare din tabelul 4.9 regăsim doar 12 microrutine de execuție pentru instrucțiunile din clasa D. Instrucțiunile de resetare a flag-urilor de condiții (instrucțiunile CLC, CLZ, CLS, CLV și CCC) au alocată o microrutină de execuție comună (vezi prima observație de la 4.1.4 și microprogramul de emulare din tabelul 4.9).

ORGANIZAREA ȘI PROIECTAREA CALCULATOARELOR

Analog, instrucțiunile de setare a *flag*-urilor de condiții (instrucțiunile SEC, SEZ, SES, SEV și SEC) au alocată o microrutină de execuție comună (vezi prima observație de la 4.1.4 și microprogramul de emulare din tabelul 4.9).

Utilizarea celor două măști codificate în codul celor 10 instrucțiuni de setare/resetare a *flag*-urilor de condiții (vezi prima observație de la 4.1.4), permite colapsarea microrutinelor de execuție aferente acestor instrucțiuni. Prin această colapsare se obține o reducere a lungimii microprogramului de emulare și desigur o reducere cantitativă a *hardware*-ului de implementare și a complexității acestuia.

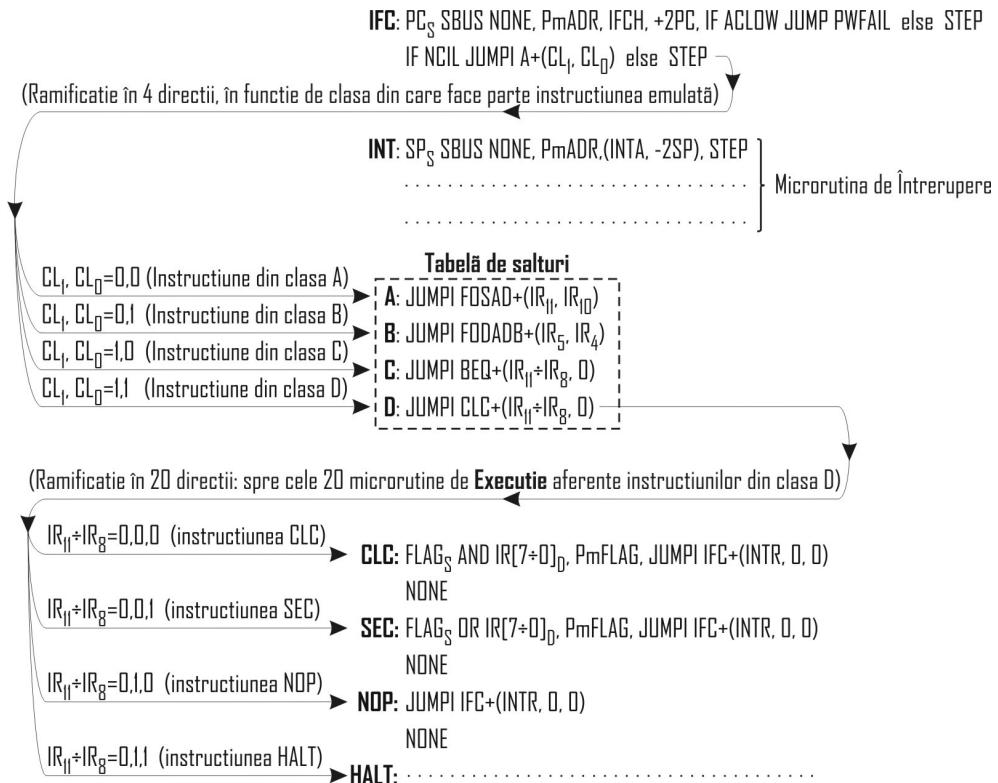


Fig 4.35 Mecanismul de selecție a microrutinelor în cadrul procesului de emulare a unei instrucțiuni din clasa D

După cum am menționat deja la instrucțiunile din clasa A, toate microrutinile de execuție se încheie cu punctul interuptibil, implementat tot printr-un salt indexat:

JUMPI IFC+(INTR, 0, 0)

Excepție face microrutina de execuție aferentă instrucțiunii DI (*Disable Interrupts*), unde punctul interuptibil nu are sens (vezi eticheta DI, respectiv microadresa 64_H în tabelul 4.9).