Ioan Z. Mihu

Organizarea şi Proiectarea Calculatoarelor

Concepte Fundamentale

Editura Universităţii ”Lucian Blaga” din Sibiu
2016

Ioan Z. MIHU

Organizarea şi Proiectarea Calculatoarelor

Concepte Fundamentale

Editura Universităţii ”Lucian Blaga” din Sibiu, 2016

ISBN 978-606-12-1436-5

Cuprins

1 Introducere 1

1.1 Scurt istoric 1
1.2 Structura calculatorului numeric 12
1.2.1 Procesorul 13
1.2.2 Memoria Principală 13
1.3 Formatul Instrucţiunii 15
1.3.1 O argumentaţie logică din care rezultă diverse formate 15
1.3.2 Formatul instrucţiunii la procesoarele CISC 19
1.3.3 Formatul instrucţiunii la procesoarele RISC 24
1.4 Ierarhizarea memoriei 29
1.5 Registrele generale 31
1.5.1 Registre generale implementate în structuri uniport 32
1.5.2 Registre generale implementate în structuri multiport 35
1.6 Memoria internă 37

2 Procesor didactic. Structură şi proiectare 39

2.1 Schema bloc a procesorului didactic 40
2.2 Proiectarea setului de instrucţiuni 48
2.2.1 Formatul instrucţiunilor din clasa IOP 49
2.2.2 Formatul instrucţiunilor din clasa ID 54
2.3 Ciclul instrucţiunii 59
2.3.1 Faza Fetch Instrucţiune (IF) 59
2.3.2 Faza Fetch Operand (OF) 59
2.3.3 Faza de Execuţie (EX) 60
2.3.4 Faza de Întrerupere (INT) 60
2.4 Succesiunea fazelor 64

2.5 Proiectarea fazei IF ... 65
2.5.1 Organigrama fazei IF ... 66
2.5.2 Tabelul de comenzi aferent fazei IF ... 70
2.6 Proiectarea fazei OF ... 73
2.6.1 Organigrama fazei OF ... 74
2.6.2 Tabelul de comenzi aferent fazei OF ... 76
2.7 Proiectarea fazei EX ... 78
2.7.1 Faza EX pentru instrucţiunile din clasa IOP ... 79
A. Instrucţiunile „*load*” ... 79
B. Instrucţiunea ADD A,*src* ... 80
C. Instrucţiunea SUB A,*src* ... 85
D. Instrucţiunea CMP A,*src* ... 86
E. Instrucţiunile logice ... 87
F. Instrucţiunile INC *dest* şi DEC *dest* ... 88
G. Instrucţiunile PUSH *reg* şi POP *reg* ... 93
H. Instrucţiunile „*store*” ... 95
I. Instrucţiunea JMP *adr* ... 97
J. Instrucţiunile de salt condiţionat ... 101
K. Instrucţiunea CALL *adr* ... 102
2.7.2 Faza EX pentru instrucţiunile din clasa ID ... 105
A. Instrucţiunile INC A, DEC A şi NEG A ... 105
B. Instrucţiunile PUSH A şi PUSH FLAG ... 107
C. Instrucţiunile POP A şi POP FLAG ... 108
D. Instrucţiunile de deplasare şi rotire ... 109
E. Instrucţiunile NOP şi HALT ... 116
F. Instrucţiunile EI şi DI ... 118
G. Instrucţiunile RET şi IRET ... 120
2.8 Proiectarea fazei INT ... 121
2.8.1 Organigrama fazei INT ... 122
2.8.2 Tabelul de comenzi aferent fazei INT ... 124
2.9 Implementarea în *hardware* a elementelor procesorului didactic ... 125
2.9.1 Tipuri de comenzi ... 125
2.9.2 Conectarea registrelor la busuri ... 127
2.9.3 Unitatea ALU ... 129
2.9.4 Determinarea frecvenţei semnalului de tact ... 136
2.10 Proiectarea unităţii de control ... 138
2.10.1 Generatorul de faze (GF) ... 141
2.10.2 Generatorul impulsurilor de stare (GIS) ... 143
2.10.3 Reţeaua Logică Combinaţională (RLC) ... 146
2.11 Sistemul de întreruperi şi excepţii ... 154
2.11.1 Memorarea şi prioritizarea cererilor de întrerupere ... 156
2.11.2 Generarea cererii globale de întrerupere INTR ... 159
2.11.3 Memorarea şi prioritizarea excepţiilor ... 161
2.11.4 Generarea vectorilor de întrerupere/excepţie ... 162
2.11.5 Perfecţionarea sistemului de întreruperi şi excepţii ... 166

3 Microprogramare. Concept şi implementare 173

3.1 Introducere 173
3.2 Schema bloc de principiu a unităţii de control microprogramate.....174
3.3 Fazele proiectării unui procesor microprogramat.....176
3.4 Formatul microinstrucţiunii 177
3.4.1 Formatul microinstrucţiunii de ramificaţie 178
A. Successorii JUMP μADR 182
B. Successorii JUMPI μADR+INDEX.....183
3.4.2 Formatul microinstrucţiunii de transfer.....189
A. Formatul microinstrucţunii de transfer sincron.....189
B. Formatul microinstrucţunii de transfer asincron.....196
3.5 Automatul SEQ 202
3.5.1 Organigrama de funcţionare a automatului SEQ 202
3.5.2 Graful de tranziţii al automatului SEQ.....207
3.5.3 Schema bloc a automatului SEQ.....209
3.5.2 Implementarea automatului SEQ cu memorie ROM şi registru 209
3.5.5 Evaluarea funcţională a automatului SEQ 216
3.5.6 Implementarea automatului SEQ cu bistabil asociat la fiecare stare.....228
A. Graful de tranziţii.....230
B. Ecuaţiile intrărilor D_0, D_1, D_2 şi D_3.....230
C. Ecuaţiile intrărilor D_0, D_1, D_2 şi D_3.....232
D. Schema detaliată a automatului 233
3.5.7 Implementarea automatului SEQ cu bistabili şi stări codificate 235
A. Graful de tranziţii.....237
B. Ecuaţiile intrărilor J_A, K_A, J_B şi K_B 238
C. Ecuaţiile ieşirilor.....243
D. Schema detaliată a automatului 243
3.6 Schema bloc a unităţii de control microprogramate aferentă procesdorului didactic 245
3.7 Proiectarea blocurilor din componenţa unităţii de control microprogramate 249
3.7.1 Blocul de decodificare microcomenzi.....249
A. Generarea microcomenzilor pentru selecţia sursei pe SBUS, DBUS, RBUS şi pentru ALU 249
B. Generarea microcomenzilor „Primeşte Registru”.....252
C. Generarea microcomenzilor „Other Operations”.....254
D. Generarea microcomenzilor „*Shift&Other Operations*”.....254
3.7.2 Blocul de selecţie index 255
3.7.3 Blocul de selecţie a condiţiei de ramificaţie.....257
3.7.4 Memoria de Microprogram şi registrele MAR şi MIR 259
3.8 Microprogramul de emulare.....260
3.8.1 Organigrama de principiu a microprogramului de emulare 261
3.8.2 Conţinutul microprogramului de emulare.....262
A. Microrutina Fetch Instrucţiune 276
B. Microrutinele Fetch Operand.....276
C. Microrutinele de Execuţie.....277
D. Microrutina de Întrerupere.....279

3.9 Rectificări în structura unităţii de control microprogramate 280
3.10 Înlănţuirea microrutinelor în procesul emulării instrucţiunilor 283
3.11 Minimizarea timpilor de legătură dintre microrutine 285
3.11.1 Salturi simple condiţionate 285
3.11.2 Saltul indexat JUMPI μADR+(IR_9, IR_8) 286
3.11.3 Saltul indexat JUMPI μADR+(IR_9, IR_8, 0, 0) 287
3.12 Optimizarea formatului microinstrucţiunii 289

Lista prescurtărilor utilizate 297

Bibliografie 304

Introducere

1.1 Scurt istoric

În jurul anului 3000 î.e.n., pe valea Tigrului şi Eufratului se utiliza tăbliţa cu pietricele (o tăbliţă de argilă pe care erau săpate şanţuri, în care erau aşezate pietricele). Prin deplasarea pietricelelor de-a lungul şanţurilor, număratul devenea semiautomat.

Ideea de a găuri pietricelele şi de a înşira mărgeluţele obţinute în grupuri de câte 10 a condus la creşterea substanţială a vitezei de calcul. "Calculatorul cu mărgeluţe înşirate" a fost denumit abac.

În anul 1642, la vârsta de 19 ani, Blaise Pascal a construit o maşină de adunat în care grupul de 10 mărgeluţe din structura abacului era înlocuit de o rotiţă dinţată cu 10 dinţi (câte un dinte pentru fiecare cifră zecimală).

În anul 1694, Gottfried Wilhelm Leibnitz a perfecţionat maşina de adunat astfel încât aceasta să poată efectua şi operaţii de înmultire; maşina lui Leibnitz nu mai opera în sistemul zecimal ci în sistemul binar.

În anul 1823, Charles Babbage a conceput "*Analytical Engine*", primul dispozitiv programabil, care nu a fost niciodată funcţional deoarece tehnologia de fabricaţie a părţilor componente era insuficient dezvoltată la acea vreme. Componentele principale ale maşinii "*Analytical Engine*" erau: dispozitivul de introducere (bazat pe cartele perforate), dispozitivul de memorare a numerelor în vederea prelucrării ("magazia"), dispozitivul de calcul ("moara"), dispozitivul de control şi dispozitivul de ieşire. Prin urmare, multe dintre componentele şi conceptele care au stat la baza proiectului "*Analytical Engine*" au rămas valabile şi pentru calculatoarele de astăzi. Alături de Babbage a lucrat şi Ada Augusta, contesă de Lovelace, care ar putea fi considerată primul programator. Ea a sesizat necesitatea utilizării structurilor de selecţie repetitive şi a variabilelor index. Mult mai târziu, în anul 1979, în cinstea ei va fi denumit limbajul de programare Ada.

În anul 1872, E. Barbour realizează prima maşină de calcul cu imprimantă iar în anul 1892, W. Burroughs construieşte o maşină de calcul (de birou) perfecţionată.

În anul 1912, F. Baldwin şi J. Monroe încep producţia de masă a maşinilor mecanice cu 4 operaţii aritmetice iar în anul 1930 începe producţia maşinilor electromecanice de calcul, capabile să efectueze operaţii matematice precum: adunare, scădere, înmulţire, împărţire, rădăcină pătrată, etc.

Primul calculator electromecanic programabil a fost Z1 construit de germanul Konrad Zuse în apartamentul părinţilor săi, între anii 1936-1938. Z1 avea o memorie de 64 cuvinte pe 22 biţi, era programat cu ajutorul unui cititor de bandă de hârtie perforată

(periferic de intrare) şi livra rezultatele tot pe hârtie perforată (utilizând ca periferic de ieşire perforatorul de bandă).

Primul calculator electronic digital a fost ABC (*Atanasoff-Berry Computer*) construit de către profesorul John Vincent Atanasoff împreună cu studentul Cliff Berry între anii 1937-1942 la Iowa State College (astăzi Iowa State University).

În anul 1943 începe construcţia calculatorului ENIAC (*Electronic Numerical Integrator And Computer*) proiectat la Moore School de către John Mauchly şi J. Presper Eckert, sprijiniţi fiind de către John von Neumann şi alţi colaboratori.

În anul 1944 la Harvard era lansat calculatorul electromecanic Mark I care avea la bază o serie de idei ale lui Howard Aiken şi care era destinat calculului unor tabele complexe pentru armata americană (U. S. Navy). La începutul anului 1945, când Mark I a fost oprit pentru reparaţii, Grace Hopper (unul dintre cei trei programatori care lucrau pe Mark I) nota că un şoarece prins într-un releu a cauzat probabil defecţiunea. Începând cu acea zi Hopper a introdus şi utilizat termenul "*debugging*" pentru a defini activitatea de depistare a defectelor prin proceduri specifice de testare.

În 1946 se finalizează construcţia calculatorului ENIAC. Deşi proiectul ENIAC a durat doar trei ani, la inaugurare ENIAC era tehnologic învechit. Probabil din acest motiv, înainte de finalizarea construcţiei calculatorului ENIAC, inventatorii acestuia au decis să lanseze proiectul unui calculator mai modern, EDVAC (*Electronic Discrete Variable Automatic Computer*). În directă legătură cu calculatorul EDVAC, în anul 1945 (30 iunie) von Neumann introducea conceptul calculatorului cu program memorat, într-o lucrare intitulată "Prima schiţă de raport asupra calculatorului EDVAC", lucrare care nu a fost niciodată publicată official dar care a avut un impact major (introducea "structura von Neumann" pe care o referim şi astăzi).

Programarea calculatorului ENIAC presupunea recablarea unor conexiuni; o ultimă versiune a acestei maşini elimina acest inconvenient. Cu câteva ore înainte de inaugurare, pentru a da o imagine mai impresionantă calculatorului în faţa reporterilor, unul dintre membrii echipei de proiectare (probabil Eckert) a fixat sfere translucide deasupra beculeţelor (jumătăţi de mingi de ping pong). ENIAC a fost brevetat (ulterior) de către Oficiul de Patente al Statelor Unite ca fiind primul calculator electronic.

La construcţia calculatorului ENIAC au fost utilizate aproximativ 18000 de tuburi electronice cu vid iar calculatorul cântărea aproximativ 50 de tone. Să menţionăm că după 27 de ani de la inaugurarea calculatorului ENIAC, la 19 octombrie 1973, judecătorul federal Earl R. Larson pronunţă sentinţa prin care invalidează patentul acordat de către Oficiul de Patente al Statelor Unite lui Eckert şi Maucly, pentru ENIAC. Prin aceeaşi sentinţă, Atanasoff este recunoscut ca fiind inventatorul primului calculator electronic digital (calculatorul ABC). Pentru că Eckert şi Maucly n-au făcut niciodată recurs la sentinţa pronunţată în favoarea lui Atanasoff, cred că este potrivit să considerăm că ABC a fost primul calculator electronic digital. Totuşi, pentru că ABC nu a fost niciodată complet funcţional, se poate considera că primul calculator electronic digital funcţional a fost ENIAC.

În 1947 John Bardeen, Walter Brattain şi William Shockley de la Bell Labs finalizează cercetările şi inventează tranzistorul. Pentru această invenţie, în anul 1956, cei trei au primit premiul Nobel pentru fizică. Începând cu 1948, echipe de cercetători din toată lumea lucrau la aşa zisa maşină cu program memorat (*stoted program computer*).

Calculatorul EDSAC produs în Anglia este considerat a fi primul calculator electronic cu program memorat. EDSAC a făcut primele calcule la 6 mai 1949 şi a fost calculatorul pe care a rulat primul joc graphic de calculator.

În următorii 5 ani progresele realizate în domeniul calculatoarelor s-au datorat în principal guvernelor şi armatelor, care au finanţat cele mai multe proiecte. Calculatorul UNIVAC construit de compania Remington-Rand şi livrat către Census Bureau în 1951 a cauzat pierderi financiare foarte mari constructorului. În anul următor (1952) Grace Hopper, care pe atunci era angajat al companiei Ramington-Rand, propunea reutilizarea *software*-ului (*reusable software*); se referea la segmente de cod care puteau fi extrase şi asamblate, în acord cu anumite directive (instrucţiuni), într-un limbaj de nivel mai înalt (*higher level language*). Astfel s-a născut conceptul compilării. Hopper va revizui acest concept timp de 20 de ani şi ideile sale au fost implementate în toate calculatoarele moderne. CBS a utilizat unul dintre cele 46 calculatoare UNIVAC produse, pentru predicţia rezultatului alegerilor prezidenţiale din Statele Unite, din anul 1952. Rezultatul estimării nu a fost transmis la radio timp de trei ore deoarece cei implicaţi nu au avut încredere în predicţia calculatorului (care în final s-a dovedit corectă).

În anul 1953 IBM lansează sistemul 701, primul calculator cu succes comercial incontestabil. În 1956 este lansat limbajul FORTRAN (propus încă din 1954; construcţia compilatorului a durat aproape 3 ani). Două alte limbaje, LISP şi COBOL, sunt lansate în 1957 şi respectiv 1958. ALGOL şi BASIC sunt de asemenea limbaje din prima generaţie; deşi n-a fost niciodată larg utilizat, ALGOL reprezintă baza pe care s-au construit multe dintre limbajele actuale.

O dată cu lansarea în anul 1958 de către Control Data a sistemului CDC1604, primul calculator electronic construit cu tranzistori, se naşte o nouă generaţie de calculatoare. Echipa de dezvoltare a proiectului CDC1604 era condusă de renumitul cerecetător Seymour Cray. Tot în 1958, doi cercetători (Jack Kilby şi John Noyce) care lucrau independent, lansează circuitele integrate.

În 1961 Fairchild Semiconductor produce primul circuit integrat. După 10 ani toate calculatoarele utilizau circuite integrate în loc de tranzistori. Tot în 1961 este creat limbajul de programare FORTRAN IV.

În 1962 calculatorul Atlas proiectat la universitatea din Manchester devine operaţional; în acest sistem se găseau multe dintre inovaţiile arhitecturale prezente şi în sistemele de calcul actuale: procesarea instrucţiunilor cu suprapunere de faze (*pipelineing*), paginarea memoriei, memoria virtuală (denumită iniţial „*one level store*”).

La 7 aprilie 1964 IBM lansează sistemul 360 care, pe lângă faptul că reprezenta o minune a tehnicii, a beneficiat şi de o strategie inovatoare de „business” dar şi de o orientare preponderentă către „business”: IBM garanta compatibilitatea în jos şi respectiv în sus a noului sistem reducând astfel riscul beneficiarului de a investi într-o tehnologie învechită. Unitatea centrală a sistemului 360 era microprogramată şi unul dintre motivele cele mai importante pentru utilizarea microprogramării a fost acela de a emula, într-o manieră mai eficientă, vechile sisteme IBM, pe noul 360. Asigurarea utilizatorilor privind trecerea directă a programelor de aplicaţii de pe vechile sisteme pe noile sisteme mai performante (compatibilitatea în jos) a constituit o tehnică managerială extrem de valoroasă lansată de către IBM.

Tot în 1964, la Dartmouth University, John Kemeny şi Thomas Kurtz dezvoltă limbajul BASIC (*Beginners All-purpose Symbolic Instruction Language*).

După aproximativ un an, MIT revine în top prin introducerea unei reţele care partaja resurse şi care utiliza primul minicalculator (minicalculatorul PDP-8 produs de compania DEC); era o reţea care gestiona liniile telefonice. Bell Labs şi General Electric au fost actori importanţi în acest proiect.

În 1968 Robert Noyce şi Gordon Moore, nemulţumiţi de activitatea pe care o desfăşurau la Fairchild Semiconductor, fondează Intel Corporation.

În 1969 compania AT&T Bell Laboratories, nu foarte mulţumită de direcţia în care evolua proiectul MIT, dezvoltă propriul sistem de operare, UNIX; foarte discret era lansat unul dintre principalii precursori ai sistemelor actuale de internet.

Primul calculator personal este considerat Kenback-1 care a fost lansat sub formă de kit (set de piese electronice pe care utilizatorul urma să le asambleze pentru a obţine calculatorul) în anul 1971 şi care costa 750$. Kenback-1 utiliza o serie de *switch*-uri pentru introducerea datelor în calculator şi respectiv o serie de beculeţe pentru a afişa datele de ieşire (rezultatele). Primul calculator vândut ca şi produs final (nu sub formă de kit) a fost calculatorul Micral; Micral era construit în jurul unui microprocesor Intel 8008 şi era vândut în anul 1973 la preţul de 1750$.

În anul 1971 Niklaus Wirth inventează limbajul de programare Pascal şi tot în 1971, la 15 noiembrie, Intel lansează pe piaţă primul microprocesor, 4004. Prima ediţie a sistemului de operare UNIX este lansată pe piaţă la 11 martie 1971 şi tot în 1971 apare şi prima ediţie a „Unix PROGRAMMER'S MANUAL” scrisă de K. Thompson şi D. M. Ritchie.

Tot în 1971 IBM realizează primul program de recunoaştere a vorbirii capabil să recunoască aproximativ 5000 de cuvinte din limba engleză.

În 1972 Dennis Ritchie de la Bell Labs inventează limbajul de programare C şi tot în 1972 este lansat limbajul de programare FORTRAN 66.

Tot în 1972, Ray Tomlinson introduce reţeua de e-mail, primul sistem de transmitere de mesaje capabil să transmită mesaje oricărui utilizator din cadrul unei reţele de calculatoare.

În 1973 Vinton Cerf şi Robert Kahn proiectează TCP (*Transmission Control Protocol*). După cinci ani (1978), Danny Cohen, David Reed, şi John Shoch subdivizează TCP în două protocoale distincte (care operează pe nivele diferite) rezultând astfel TCP/IP (*Transmission Control Protocol / Internet Protocol*). IP utilizează adrese de pachete şi permite fiecărui calculator conectat la internet să *forward*-eze un anumit pachet altui calculator care este cu un pas (sau mai mulţi paşi) mai aproape de destinatarul pachetului. TCP asigură corectitudinea transmisiei (verifică erorile de transmitere şi emite cereri de retransmisie dacă detectează erori); suplimentar TCP recompune (la destinatar) în ordinea corectă pachetele din cadrul unui mesaj

Deşi nu a fost niciodată vândut, calculatorul Alto lansat de compania Xerox în anul 1974 este considerat primul sistem de tip “*workstation*”. Alto era un sistem revoluţionar pentru acea perioadă deoarece conţinea un calculator complet operaţional, un *display* şi un *mouse*. Alto opera ca multe dintre calculatoarele de astăzi: utiliza ferestre, meniuri şi *icon*-uri pentru interfaţarea cu sistemul de operare.

Tot în 1974 Intel lansează microprocesorul 8080 (pe 8 biţi) care va deveni practic un standard în industria calculatoarelor; după 2 ani (în 1976), Intel va lansa microprocesorul 8086 (pe 16 biţi).

În anul 1975 Ed Roberts, o dată cu lansarea calculatorului Altair 8800 care dispunea de o memorie de un kilobyte şi care era vândut sub formă de kit, introduce şi sintagma “*personal computer*”. Bill Gates, Paul Allen, şi Monte Davidoff scriu primul compilator de BASIC pentru această maşină (Altair BASIC). În acelaşi an, Bill Gates renunţă la Harvard pentru a fonda Microsoft împreună cu Paul Allen.

În 1976 Microsoft lansează o versiune îmbunătăţită a limbajului BASIC. În acelaşi an (1976) Steve Wozniak proiectează primul calculator din familia Apple, Apple I. Mai apoi, Steve Wozniak împreună cu Steve Jobs înfiinţează compania Apple Computers. Tot în 1976 Intel lansează microprocesorul 8086 (pe 16 biţi).

În următorii ani calculatoarele personale “explodează” pe piaţa americană. Se înfiinţează foarte multe companii implicate puternic în producţia de *hardware* şi respectiv *software* destinată calculatoarelor personale. Începând cu 1977 calculatoarele personale încep să fie vândute şi în magazine. Urmează o perioadă cu multiple perfecţionări arhitecturale şi tehnologice care au făcut ca preţul şi dimensiunea calculatorului personal să scadă continuu iar putera de calcul a acestuia să crească sistematic.

În 1977 Apple Computers lansează calculatorul Apple II, primul calculator personal cu monitor color iar în iunie 1978 Apple lansează Apple DOS 3.1, primul sistem de operare destinat calculatoarelor Apple. Tot în 1978 Microsoft lansează o nouă versiune a limbajului COBOL.

În 1979 Intel lansează microprocesorul 8088 (1 iulie), Oracle (fondată în acelaşi an) lansează prima versiune comercială a SQL şi limbajul de progtramare DoD-1 este oficial transformat în Ada. Tot în 1979 este lansat microprocesorul Motorola 68000 (care ulterior va fi ales ca procesor pentru calculatoarele Apple Macintosh) şi tot în 1979 DEC (*Digital Equipment Corporation*) lansează sistemul de operare (*multi-user-multitasking*) VMS (*Virtual Memory System*) dedicat calculatoarelor VAX şi Alpha (produse de către DEC).

În 1980 IBM “angajează” pe Paul Allen şi Bill Gates pentru a crea un sistem de operare destinat unui nou calculator personal (care urma să fie lansat pe piaţă de către IBM). Cei doi cumpără un sistem de operare simpu produs de *Seattle Computer Products* şi folosesc acest sistem ca şablon. IBM permite celor doi să păstreze drepturile de comercializare a noului produs, denumit DOS (*Disk Operating System*). Tot în 1980 IBM contractează Microsoft pentru dezvoltarea unor noi versiuni de BASIC, FORTRAN, COBOL şi PASCAL destinate calculatoarelor personale ce urmau a fi produse de IBM. Astfel, după o perioadă de ezitări (abia în 1981), IBM va lansa pe piaţă primul *IBM Personal Computer*, care rula sistemul de operare MS-DOS (*Microsoft Disk Operating System*) şi care a avut un preţ de start de 1565 $.

În anul 1981 Xerox lansează staţia grafică Star, calculator care va influenţa puternic evoluţia ulterioară a calculatoarelor Lisa şi Mcintosh produse după 1981 de Apple, precum şi evoluţia sistemelor de operare Microsoft Windows. Tot în 1981 Adam Osborne lansează Osborne I, primul calculator portabil de succes, care cântărea aproximariv 11 Kg.

În anul 1982:

- Intel lansează procesorul 80286 (la 1 februarie).
- WordPerfect Corporation lansează WordPerfect 1.0 care va deveni unul dintre cele mai populare programe de procesare de text de pe piaţă.
- Rod Canion şi alţi ingineri de la Texas Instruments fondează Compaq Computer Corporation, companie care va produce prima clonă a calculatorului IBM PC (calculatorul portabil Compaq lansat în 1983) şi care va deveni cel mai mare *challenger* pe piaţa corporativă pentru IBM.
- Este lansat Calculatorul Commodore 64, cu 64 KB de memorie RAM, cu Microsoft BASIC şi cu un preţ care a „căzut” de la 600$ şi s-a stabilizat la 200$, devenind cel mai bine vândut calculator din toate timpurile.
- Microsoft lansează sistemul de operare MS-DOS versiunea 1.25
- Apple Computer devine prima companie producătoare de calculatoare care incasează 1 miliard de dolari pe an din vânzări
- Rich Skrenta, un elev de liceu în vârstă de 15 ani, crează primul virus de calculator cunoscut sub numele de *The Elk Cloner*.

În anul 1983:

- La 8 martie IBM lansează calculatorul personal XT (*eXtended Technology*). XT reprezenta de fapt o altă denumire pentru calculatorul IBM 5160, primul calculator IBM care conţinea un *hard disk* de 10 MB (opţional 20 MB), o unitate de *floppy* de 5 1/4", până la 640 KB de memorie RAM, utiliza procesorul Intel 8088, avea un slot ISA pe 8 biţi şi un port serial.
- Compaq lansează „Compaq Portable”, primul calculator 100% compatibil IBM-PC.
- Microsoft lansează sistemul de operare MS-DOS 2.0 în luna martie şi anunţă Microsoft Windows în luna noiembrie.

În anul 1984:

- IBM lansează modelul AT (*Advanced Technology*); modelul de bază conţinea un procesor Intel80286, 256 KB (opţional 512 KB) de memorie RAM, o unitate de *hard disk* de 20 MB şi o unitate de *floppy disk* de 1,2 MB. Prin urmare, marea diferenţă între modelul AT şi modelul XT era dată de procesorul utilizat.
- Apple Computers lansează calculatorul Apple Macintosh, cu interfaţă grafică pentru utilizatori; în 6 luni sunt vândute peste 100.000 de calculatoare.
- Microsoft lansează sistemele de operare MS-DOS 3.0 pentru calculatoarele IBM-PC AT şi respectiv MS-DOS 3.1 pentru reţele.
- Este lansat calculatorul Tandy 1000 care devine cel mai bine vândut calculator în anul 1984.

În anul 1985:

- Microsoft şi IBM încep colaborarea pentru noua generaţie de sisteme de operare (OS/2).
- Intel lansează microprocesorul 80386 în luna octombrie.
- Microsoft lansează, în luna noiembrie, sistemul de operare Windows 1.0 (vândut iniţial cu 100$)

În anul 1986:

- Compaq lansează primul calculator PC bazat pe procesorul Intel 80386.
- Microsoft este listată la bursă, acţiune care îl face pe Bill Gates unul dintre cei mai tineri miliardari ai lumii.
- Microsoft lansează sistemul de operare MS-DOS 3.2 (în luna aprilie).
- IBM lansează primul laptop, compatibil PC, care cântărea aproximativ 5,5 Kg.

În anul 1987:

- CompuServe introduce standardul GIF pentru imagini.
- IBM lansează calculatorul personal PS/2 (cu grafică îmbunătăţită, cu unitate de floppy de 3,5 inch, cu port paralel bidirecţional pe 8 biţi) şi tot IBM introduce standardul VGA (*Video Graphics Array*) pentru monitoare color (16 culori afişabile simultan, rezoluţie 640×480 pixeli).
- Microsoft lansează sistemele de operare MS-DOS 3.3 (în luna aprilie) şi Windows 2.0 (pe 9 decembrie).
- Sun lansează procesorul SPARC.

În anul 1988:

- Friedrich Reintzer descoperă cristalele lichide utilizate astăzi la construcţia monitoarelor LCD (*Liquid Crystal Displays*).
- Este introdus protocolul SNMP (*Simple Network Management Protocol*), protocol de tip interogare-răspuns-comandă care pemite examinarea şi modificarea parametrilor de configurare aferenţi dispozitivelor de interconectare (repetoare, *router*-e, *switch*-uri *brdge*-uri) din reţelele LAN şi respectiv WAN.
- Microsoft lansează sistemele de operare MS-DOS 4.0 (în luna iulie) şi MS-DOS 4.1 (în luna noiembrie).
- Motorola lansează microprocesorul 88000 (procesor RISC pe 32 de biţi utilizat de Apple la construcţia calculatorului Power PC).
- Intel lansează microprocesorul 80386SX.

În anul 1989 Intel lansează microprocesorul 80486DX care conţinea peste un milion de tranzistoare şi care era dotat cu resurse pentru *multitasking*.

În anul 1990:

- Tim Berners-Lee, care lucra împreună cu Robert Cailliau la CERN, propune un sistem „*hypertext*” care va reprezenta un prim pas către internetul de astăzi.
- Microsoft lansează sistemul de operare Windows 3.0, o versiune de Windows complet nouă şi care a fost vândută în peste 3 milioane de copii în primul an. Tot Microsoft, la capitolul vânzări, depăşeşte cifra de un miliard de dolari pe an, devenind prima companie care atinge acest prag.
- Microsoft şi IBM decid să înceteze colaborarea privind dezvoltarea sistemelor de operare.
- IBM lansează XGA (*eXtended Graphics Array*), standard video cu 256 culori (simultane) şi rezoluţia de 1024×768 pixeli.
- Intel lansează microprocesorul 80386SL, proiectat pentru consum mic de putere (*low power*) şi care va fi utilizat în multe calculatoare portabile.
- Este lansată prima maşină de căutare (Archie), scrisă de Alan Emtage, Bill Heelan şi Mike Parker de la McGill University din Montreal, Canada

În anul 1991:

- Tim Berners-Lee lansează WWW (*World Wide Web*) către marele public (la 6 august). WWW reprezintă o interfaţă grafică pentru internet, care furnizează un mediu extrem de bogat de documente (conţinând text, fişiere grafice, fişiere multimedia, etc.), cu legături între ele şi care sunt puse la dispoziţia utilizatorilor de internet.
- AMD lansează microprocesorul AM386 (în luna martie).
- Intel lansează (în aprilie) microprocesorul 80486SX, în ideea de a oferi producătorilor de calculatoare un procesor mai ieftin (258$).
- Linus Torvald lansează sistemul de operare LINUX.
- Este lansat limbajul de programare FORTRAN 90.
- Cosecvent deciziei anterioare de a nu mai dezvolta sisteme de operare în colaborare cu IBM, Microsoft schimbă denumirea OS/2 în Windows NT.
- Microsoft lansează sistemul de operare MS-DOS 5.0 (în luna iunie).

În anul 1992:

- Microsoft lansează sistemul de operare Windows 3.1 care va fi vândut în peste un milion de copii în primele două luni de după lansare.
- Intel lansează (la 2 martie) microprocesorul 80486DX2 care avea o opţiune de dublare a frecvenţei semnalului de tact, ceea ce conducea la o viteză semnificativ mai mare de operare.
- IBM lansează ThinkPad, primul *notebook* care dispunea de un ecran TFT (*Thin Film Transistor = display* cu cristale lichide de tip *active matrix*) color de 10,4 *inch* şi de *TrackPoint* (*mouse* implementat sub forma unui mic *joystik* izometric plasat pe tastatură între tastele "G", "H" şi "B").
- SQL Access Group lansează ODBC (*Open Database Connectivity*).

În anul 1993:

- Microsoft lansează Windows NT 3.1, Microsoft Office 4.0 şi MS-DOS 6.0.
- Intel lansează (la 22 martie) microprocesorul Pentium care incorpora peste 3,2 milioane de tranzistoare, avea o frecvenţă a semnalului de tact de 60 MHz şi costa 878$.
- IBM, Motorola şi Apple lansează microprocesorul PowerPC care va fi ulterior utilizat în construcţia calculatoarelor Apple Power Mac.

În anul 1994:

- Microsoft lansează sistemele de operare Windows 95 -versiunea beta, MS-DOS 6.22, Windows 3.11 şi Windows NT 3.5.
- Intel lansează (la 7 martie) a doua generaţie de procesoare Intel Pentium.
- Un profesor de matematică descoperă, la procesorul Pentium, unul dintre cele mai faimoase *bug*-uri *hardware* (rezultate incorecte generate de unitatea de virgulă flotantă pentru anumite configuraţii ale operanzilor). Erorile erau prezente la procesoarele Pentium cu frecvenţe de tact egale sau mai mici de 100 MHz; erorile nu mai erau prezente la procesoarele cu frecvenţe de tact egale sau mai mari de 120 MHz. Compania Intel a fost determinată să retragă şi să înlocuiască milioane de procesoare.
- David Filo şi Jerry Yang lansează (în luna aprilie) YAHOO (*Yet Another Hierarchical Officious Oracle!*).

În anul 1995:

- Este lansat primul *software* de tip VoIP (Vocaltec) care permitea utilizatorilor finali să facă apeluri prin voce (*voice calls*) pe Internet
- Microsoft lansează Windows NT 3.51 (în data de 30 mai), Internet Explorer 1.0 (în august) şi Windows 95 care în primele 4 zile s-a vândut în peste 1 milion de exemplare.
- Sun lansează limbajul de programare Java şi LiveScript este redenumit JavaScript.
- Intel lansează (în noiembrie) procesorul Intel Pentium Pro.
- Larry Page şi Sergey Brin dezvoltă PageRank (aplicaţie de sortare a paginilor web printr-un algoritm care acordă fiecărei pagini o pondere numerică). Ulterior PageRank va fi incorporat în motorul de căutare Google pentru sortarea (ierarhizarea) rezultatelor căutărilor efectuate de către utilizatori.

În anul 1996:

- Google este fondat şi lansat oficial (la 7 septembrie) de către Sergey Brin şi Larry Page.
- Microsoft lansează Windows NT 4.0 (la 29 iulie), Internet Explorer 3.0 (în august) şi Microsoft Windows CE 1.0 (ca soluţie de sistem de operare portabil).
- Intel lansează procesorul P6.

În anul 1997:

- Microsoft lansează Internet Explorer 3.0 pentru calculatoarele Apple Macintosh (la 7 ianuarie), Internet Explorer 4.0 (în septembrie) şi Microsoft Office 97 .
- Intel lansează procesorul MMX (*MultiMedia eXtension*); cu cele 57 instrucţiuni noi, introduse pentru creşterea vitezei de procesare a aplicaţiilor grafice şi respectiv multimedia, MMX-ul este acum inclus în toate procesoarele Intel şi AMD (*Advanced Micro Devices*). Tot în 1997 (la 7 mai), Intel lansează procesorul Pentium II.

În anul 1998:

- Microsoft lansează sistemul de operare Microsoft Windows 98 (la 25 iunie).
- Intel lansează procesorul Celeron.
- Este lansat sistemul de management al bazelor de date relaţionale MySQL, care va deveni o alternativă foarte populară (*open source* şi *free*) în raport cu alte soluţii SQL.

În anul 1999:

- Microsoft lansează Windows CE 3.0 şi Internet Explorer 5.0 (la 18 martie).
- Intel lansează procesorul Pentium III (la 26 februarie).

În anul 2000:

- Microsoft lansează, la 17 februarie, sistemele de operare Windows 2000 (următoarea versiune de Windows NT anunţată încă din anul 1998) şi Windows Millenium (la 19 iunie). Tot în luna iunie, Microsoft lansează către marele public C# iar la 9 noiembrie lansează DirectX.
- Jack St. Clair Kilby de la Texas Instruments (cel care a conceput şi a construit primul circuit integrat) primeşte premiul Nobel pentru fizică.

În anul 2001:

- Linus Torvalds lansează Linux Kernel (cod sursă), versiunea 2.4 (la 4 ianuarie).
- Microsoft lansează Internet Explorer 6.0 (la 27 august), Microsoft Windows XP (la 25 octombrie) şi Microsoft Windows XP 64-Bit Edition pentru sistemele bazate pe procesoare Itanium.
- Apple lansaeză iPod -ul.

În anul 2002:

- Apple lansează sistemul de operare Mac OS X 10.2.
- Microsoft lansează DirectX 9 (la 19 decembrie).

În anul 2003:

- Intel lansează procesorul Pentium M (*Pentium Mobile*) destinat calculatoarelor portabile.
- Microsoft lansează Windows Server 2003, Windows XP 64-Bit Edition pentru sistemele Itanium 2 (la 28 martie) şi Windows XP Media Center Edition 2003 (la 18 decmbrie).

În anul 2004:

- Microsoft lansează Windows XP Media Center Edition 2005 (la 12 octombrie).
- Mozilla lansează Firefox 1.0 (o alternativă din ce în ce mai credibilă la Internet Explorer 6.0).

În anul 2005 Microsoft lansează sistemul de operare Windows XP Professional ×64 Edition (la 24 aprilie) şi tot Microsoft anunţă că următorul sistem de operare (cu numele de cod "Longhorn") se va numi Windows Vista.

În anul 2006:

- Intel lansează (la 5 ianuarie) procesorul Intel Core (succesorul procesorului Pentium M), procesorul dual Core 2 Duo (la 27 iulie) şi procesorul dual Core 2 Extreme (la 29 iulie).
- Microsoft lansează (la 30 noiembrie) sistemul de operare Windows Vista (către corporaţii).

În anul 2007:

- Microsoft lansează (la 30 ianuarie) Microsoft Windows Vista şi Office 2007 (către marele public).
- Apple lansează Apple iPhone (la 29 iunie).

În anul 2008 (la 28 august) Apple lansează o nouă versiune pentru calculatoarele iMac.

Tot în 2008 (la 19 februarie), când Toshiba anunţă oficial că renunţă la fabricarea unităţilor HD-DVD (*High Definition Digital Versatile Disk*), se încheie în favoarea lui BD (*Blu-ray Disk*) „războiul” pe tema formatului dintre BD şi HD-DVD. BD a fost lansat la 4 ianuarie 2006 de un consorţiu de companii (Dell, Hitachi, Hewlett Packard, LG, Mitsubishi, Panasonic, Sony, TDK, Apple, Philips, Sony, Sun) şi este o unitate de disc optic cu capacitatea de 25 GB. HD-DVD a fost tot o unitate optică, cu capacitatea de 15 GB, lansată la 31 martie 2006 de un alt consorţiu (Toshiba şi NEC), în ideea de a înlocui unitatea clasică de DVD.

În aprilie 2009, General Electric anunţă inventarea discului holografic cu capacitatea de 500 GB. Discul holografic va avea un format similar DVD-ului, astfel că noile cititoare care vor fi create pentru discul holografic vor putea citi şi formatele DVD şi Blue-Ray. Se estimează că discul holografic va fi lansat în cursul anului 2011.

Privind retrospectiv spectaculoasa evoluţie a calculatoarelor electronice, am putea defini şase generaţii (tabelul 1.1):

Generaţii	Caracteristici Tehnologice	Performanţe	Limbaje de Programare
Generaţia I (1937–1957)	-tuburi electronice cu vid -un singur dispozitiv de memorare (tambur magnetic) de capacitate mică şi cu timp de acces mare	-viteză de calcul: 10^4 operaţii/sec. -memorie: 2 KB	Programare în: -cod maşină -limbaje de asamblare
Generaţia II (1958–1963)	-diode şi tranzistoare -memorie internă cu inele de ferită -memorie externă: tambur, disc sau bandă magnetică -unităţi de virgulă flotantă -regiştre de index	-viteză de calcul: 2×10^5 operaţii/sec. -memorie: 32 KB	-limbaje de nivel înalt: Fortran (1956) Algol (1958) Cobol (1959)
Generaţia III (1964–1982)	-circuite integrate -memorie internă semiconductoare -memorie externă: discuri magnetice -microprogramarea, ca tehnică de proiectare a procesoarelor complexe -memoria virtuală	-viteză de calcul: 5×10^6 operaţii/sec. -memorie: 2 MB	-sisteme de operare evoluate cu "*time sharing*" (UNIX) -limbaje de nivel înalt: Pascal (1971) C (1972)
Generaţia IV (1982–1990)	-circuite integrate pe scară largă (VLSI) -discuri optice -dezvoltarea reţelelor de calculatoare	-viteză de calcul: 3×10^7 operaţii/sec. -memorie: 16 MB	-limbaje orientate obiect (C++) -limbaje funcţionale (LISP) -limbaje de programare logică (Prolog)
Generaţia V (după 1990)	-circuite integrate pe scară ultra largă (ULSI) -revitalizarea arhitecturilor paralele -dezvoltarea sistemelor multiprocesor	-viteză de calcul: 10^{12} operaţii/sec. -memorie: 256 MB	- limbaje concurente -sisteme *software* cu inteligenţă artificială
Generaţia VI	Ce caracteristici vor avea calculatoarele din generaţia VI ? Probabil vor avea performanţe asemănătoare cu cele ale calculatorului HAL9000, din romanul lui *Arthur C. Clarke*, "*2001: A Space Odyssey*". HAL9000 putea întreţine conversaţii cu operatorii umani, putea accepta intrări vizuale, putea învăţa din propria experienţă. HAL9000 reprezintă o ficţiune dar calculatoarele din generaţia VI vor avea cu siguranţă capabilităţiule lui HAL9000; vor emula inteligenţa şi raţionamentul uman, vor dispune de interfeţe inteligente pentru comunicarea cât mai naturală (verbală şi vizuală) cu operatorul uman, vor fi capabile să înveţe şi să acumuleze experienţă.		

Tabelul 1.1 Evoluţia calculatoarelor electronice - o clasificare în şase generaţii

1.2 Structura calculatorului numeric

Un sistem de calcul (fig. 1.1.) conţine trei blocuri esenţiale în structura sa:

1. Procesorul (P) denumit uneori şi unitate centrală (UC).
2. Memoria Principală (MP) denumită uneori şi memorie centrală (MC) sau internă (MI).
3. Blocul de intrare-ieşire (I/E).

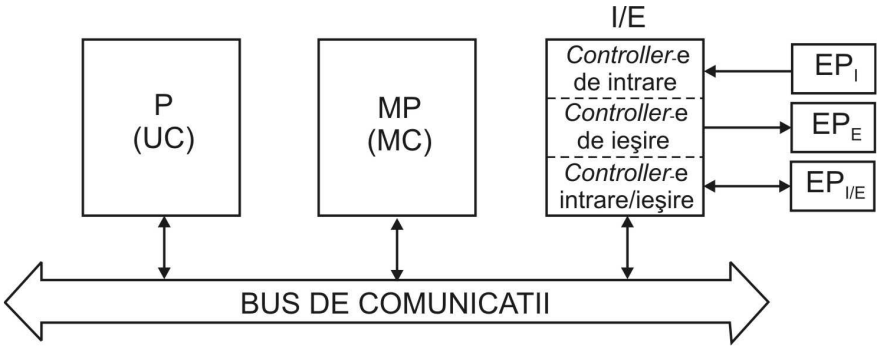


Fig. 1.1 Schema bloc a calculatorului numeric.

Fiecare dintre unităţile evidenţiate în fig. 1.1. este capabilă să execute anumite lucrări specifice din cadrul sarcinii globale ce revine întregului sistem:

- Procesorul citeşte şi execută secvenţial, una câte una, instrucţiunile unui program stocat în MP.

- MP memorează şi ţine la dispoziţia procesorului instrucţiuni (programe), date, rezultate.

- Blocul I/E conţine interfeţe de intrare-ieşire dedicate diverselor echipamente periferice conectate la sistem (EP_I/EP_E/$EP_{I/E}$ – echipamente periferice de intrare/ieşire/intrare-ieşire). Fiecare periferic va avea propria interfaţă I/E (I/O *controller*).
  Principalele echipamente periferice care apar în configuraţia unui sistem de calcul sunt:
  - EP_I: tastatură, *mouse*, *scanner*, unitate CD-ROM (echipamente periferice de intrare).
  - EP_E: imprimantă, *display* (echipamente periferice de ieşire).
  - $EP_{I/E}$: disc *hard*, disc *floppy*, unitate CD-*writer* (echipamente periferice de intrare-ieşire).

1.2.1 Procesorul

Procesorul execută în manieră secvenţială (una câte una) instrucţiunile unui program stocat în memorie (MP). Pentru procesarea unei instrucţiuni procesorul parcurge (în timp) două faze succesive:

1. ***Fetch*-ul instrucţiunii** (extragerea insrucţiunii din memorie şi încărcarea acesteia în procesor, într-un registru special denumit *Instruction Register*). Imediat după încărcarea instrucţiunii în *Instruction Register* (IR), aceasta va fi decodificată.

2. **Execuţia instrucţiunii** (numai după decodificarea instrucţiunii procesorul „află" efectiv despre ce instrucţiune este vorba şi evident va putea trece la execuţia acesteia).

Prin urmare, un program este format dintr-o succesiune de instrucţiuni. Înainte de a fi lansat în execuţie, programul trebuie încărcat în memoria principală (MP). De aici programul va putea fi executat, secvenţial (instrucţiune cu instrucţiune) şi pentru fiecare instrucţiune procesorul parcurge (în timp) cele două faze succesive: ***fetch*** şi respectiv **execuţie**. După execuţia instrucţiunii curente, procesorul trece întotdeauna la *fetch*-ul instrucţiunii următoare. Execuţia programului se încheie o dată cu execuţia ultimei sale instrucţiuni.

1.2.2 Memoria Principală

Memoria Principală este implementată fizic cu circuite RAM. Sistemele actuale utilizează circuite de tip RAM-dinamic deoarece celula de memorie RAM-dinamic are o structură foarte simplă, ceea ce permite realizarea unor circuite de memorie de capacitate mare. Capacitatea mare de memorare a circuitelor RAM-dinamic şi preţul relativ scăzut per bit memorat reprezintă avantaje nete care contrabalansează dezavantajul reprezentat de necesitatea regenerării periodice a informaţiilor memorate.

Din punctul de vedere al structurii sale logice, MP trebuie privită ca o stivă de locaţii de memorie (fig. 1.2.) în care se depozitează informaţii. Un grup de **m** celule asociate formează o locaţie de **m** biţi; **m** este de regulă o putere a lui 2. De exemplu, într-un IBM-PC care are unitatea centrală implementată cu un procesor INTEL Pentium (procesor pe 32 biţi), locaţiile de memorie (MP) vor fi pe 32 biţi (**m**=32).

Fiecare celulă memorează un bit. O locaţie formată din **m** celule memorează un cuvânt de **m** biţi. Fiecărei locaţii îi este asociată o adresă prin intermediul căreia procesorul poate accesa locaţia respectivă pentru scriere sau citire.

Asupra unei locaţii de memorie procesorul poate executa două operaţii:

a) *READ* (citire);
b) *WRITE* (scriere).

Ambele operaţii comportă două etape succesive:

1. Selecţia locaţiei. Procesorul emite spre MP adresa locaţiei ce urmează a fi accesată, iar la nivelul MP adresa este decodificată şi locaţia respectivă selectată.

2. Transferul propriu-zis de date. Pentru citire procesorul va activa o comandă de tip "*read memory*" prin care datele conţinute în locaţia accesată vor fi citite şi încărcate în procesor (într-un registru). Pentru scriere procesorul va activa un semnal de comandă de tip "*write memory*" şi, suplimentar, va transmite spre MP şi datele de scris. Acestea vor fi memorate (scrise) în locaţia selectată.

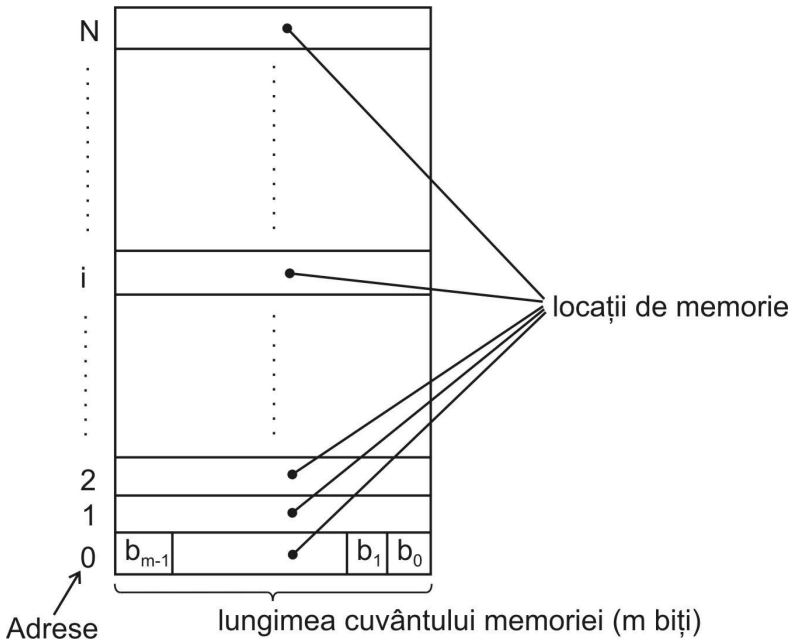


Fig. 1.2 Organizarea logică a memoriei principale

Observaţie: *Unitatea adresabilă în memorie este de regulă octetul (BYTE-ul). În consecinţă, fiecare octet va avea asignată propria-i adresă şi pentru octeţi succesivi adresele cresc din 1 în 1. Dacă, de exemplu, memoria este organizată în locaţii de 32 biţi (deoarece calculatorul este construit în jurul unui procesor pe 32 biţi) atunci, fiecare locaţie va conţine 4 octeţi şi pentru o succesiune de locaţii adiacente adresele vor creşte din 4 în 4. În cazul IBM-PC, de exemplu, 2 octeţi consecutivi formează un WORD (16 biţi) şi 4 octeţi consecutivi formează un DWORD (double word=32 biţi). Instrucţiunile pot referi operanzi pe BYTE (8 biţi), pe WORD (16 biţi) sau pe DWORD (32 biţi). Prin urmare, pentru octeţi (BYTES) succesivi adresele cresc din 1 în 1, pentru WORD-uri succesive adresele cresc din 2 în 2 iar pentru DWORD-uri succesive adresele cresc din 4 în 4.*

1.3 Formatul Instrucţiunii

1.3.1 O argumentaţie logică din care rezultă diverse formate

Codul unei instrucţiuni este reprezentat de un cuvânt binar de o anumita lungime. La procesoarele RISC se utilizează de regulă formatul de lungime fixă (lungimea codului este aceeaşi pentru toate instrucţiunile) în timp ce la procesoarele CISC se ajunge de regulă la format de lungime variabilă (lungimea codului poate să difere de la o instrucţiune la alta). Formatul de lungime fixă conduce la o simplificare a decodificatorului instrucţiunii (avantaj). Formatul de lungime variabilă revendică un decodificator mai complex ceea ce conduce la creşterea semnificativă a timpului de decodificare a codului instrucţiunii. Prin urmare, formatul de lungime variabilă prezintă dezavantaje (în raport cu formatul de lungime fixă) dar trebuie înţeles că lungimea variabilă reprezintă o consecinţă inevitabilă în cazul arhitecturilor CISC, nu o caracteristică dorită.

Un program este constituit dintr-o succesiune de instrucţiuni. Înainte de lansarea în execuţie, programul (codul obiect) trebuie încărcat în memoria internă a calculatorului. Doar după încărcare, programul poate fi lansat în execuţie începând cu prima sa instrucţiune. Instrucţiunile programului, reprezentate în cod maşină, sunt stocate în locaţii succesive de memorie, succesiunea locaţiilor corespunzând în general ordinii în care instrucţiunile vor fi procesate. Adică, instrucţiunea ce urmează a se procesa se află în locaţia de memorie imediat următoare (la adresa imediat următoare). Exceptie de la această regulă apare doar la instrucţiunile de salt.

Timpul în care o instrucţiune este procesată complet se numeşte <u>ciclul instrucţiunii</u>. Ciclul instrucţiunii cuprinde două faze (subcicluri) majore:

- faza de ***fetch* instrucţiune** (*instruction fetch*).
- faza de **execuţie** (*execution*)

În faza de *fetch*, codul instrucţiunii este citit (extras) din memorie şi încărcat în procesor într-un registru special numit registrul instrucţiunii (IR - *Instruction Register*). Ieşirile bistabililor ce compun registrul IR sunt aplicate pe intrările decodificatorului instrucţiunii. Prin decodificarea codului, unitatea de control a procesorului „află” ce instrucţiune trebuie să fie executată.

În faza de execuţie se execută efectiv operaţia specificată (operaţiile specificate) de instrucţiune. La încheierea fazei de execuţie aferentă instrucţiunii curente se trece la faza de *fetch* aferentă instrucţiunii următoare. În această manieră, instrucţiunile programului sunt procesate secvenţial, una câte una.

O instrucţiune extrasă din memorie şi încărcată în procesor pentru a fi decodificată şi executată, trebuie să furnizeze unităţii de control următoarele informaţii:

- ce operaţie trebuie aplicată operanzilor (operandului, dacă instrucţiunea are un singur operand)
- adresele locaţiilor de memorie unde se găsesc cei doi operanzi (adresa locaţiei unde se găseşte operandul dacă instrucţiunea are un singur operand)
- adresa locaţiei de memorie unde trebuie depus rezultatul
- adresa locaţiei de memorie de unde trebuie extras codul următoarei instrucţiuni de executat (pentru a putea realiza *fetch*-ul următoarei instrucţiuni)

Toate aceste informaţii trebuie să fie codificate (integrate) în codul instrucţiunii. Pe baza acestui raţionament rezultă formatul cu patru adrese (figura 1.3)

	Adresă operand 1	Adresă operand 2	Adresă rezultat	Adresă următoare instrucțiune
OPCODE	Adresa1	Adresa2	Adresa3	Adresa4

Fig 1.3 Formatul cu patru adrese

În cazul formatului cu patru adrese, în codul instrucţiunii apar cinci câmpuri:

OPCODE – *OPeration CODE* (codul operaţiei). În acest câmp de biţi se codifică operaţia pe care trebuie să o efectueze instrucţiunea: adunare, scădere, *OR* (operaţia logică SAU), *AND* (operaţia logică SI), deplasare (logică sau aritmetică, la dreapta sau la stânga), salt la o anumită adresă, etc.

În următoarele patru câmpuri sunt memorate patru adrese:

- adresele celor doi operanzi (in cazul instrucţiunilor cu un singur operand se va utiliza doar adresa operandului 1, adresa operandului 2 va fi ignorată deoarece este nenecesară).
- adresa rezultatului cu ajutorul căreia se va selecta locaţia de memorie unde va fi depus rezultatul.
- adresa următoarei instrucţiuni cu ajutorul căreia se va selecta locaţia de memorie de unde va fi extras codul următoarei instrucţiuni de executat (în faza de *fetch* a urmăoarei instrucţiuni din cadrul programului).

Marele dezavataj al formatului cu patru adrese este lungimea sa excesivă (datorită celor patru adrese codul instrucţiunii se extinde pe un număr mare de biţi). Din acest motiv formatul cu patru adrese nu este utilizat în practică, el având o relevanţă exclusiv teoretică.

Simplificarea formatului se poate realiza prin reducerea numărului de adrese din componenţa codului instrucţiunii. Prima simplificare se poate face prin eliminarea câmpului **Adresa4** şi o uşoară ajustare a semnificaţiei câmpului **Adresa3**.

Se pleacă de la observaţia că instrucţiunile unui program sunt memorate le adrese succesive de memorie iar succesiunea în care instrucţiunile sunt extrase din memorie pentru a fi executate coincide în general cu succesiunea locaţiilor în care (adreselor la care) acestea sunt memorate. Cu alte cuvinte, următoarea instrucţiune de procesat se găseşte în locaţia (la adresa) următoare. Din acest motiv, pentru a o localiza se poate folosi un registru (numărător) numit *Program Counter* (*PC*) sau *Instruction Pointer* (*IP*), implementat în procesor. În timpul execuţiei instrucţiunii curente, registrul *PC* este automat incrementat. Prin incrementare, în *PC*, se obţine adresa următoarei instrucţiuni; În momentul în care faza de execuţie a instrucţiunii curente se încheie, procesorul adresează memoria cu adresa din

PC şi activează o comandă de citire din memorie. Prin urmare, va fi citit (extras) din memorie şi încărcat în procesor codul următoarei instrucţiuni (faza *fetch* instrucţiune).

Să mai notăm că, dacă codul instrucţiunii se întinde pe k octeţi, atunci incrementarea registrului *PC* trebuie făcută cu constanta k:

$$PC \leftarrow PC+k$$

Dacă inainte de incrementare în PC se află adresa instrucţiunii curente, după incrementare în PC vom avea dresa instrucţiunii utmătoare.

Introducând în structura procesorului registrul PC care, în timpul procesării instrucţiunilor unui program, va furniza întotdeauna adresa următoarei instrucţiuni de extras din memorie, se poate recurge la formatul cu trei adrese (figura 1.4).

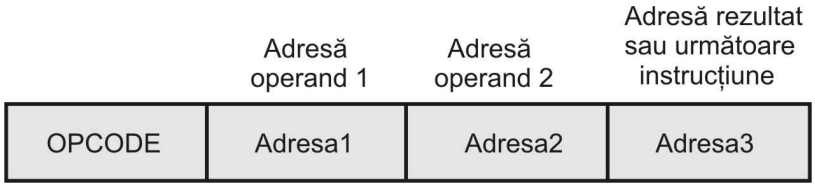


Fig 1.4 Formatul cu trei adrese

Semnificaţia generică a unei astfel de instrucţiuni ar fi: „*Extrage operanzii de la primele două adrese, execută operaţia şi depune rezultatul la* ***Adresa3*** *şi treci la fetch-ul următoarei instrucţiuni din cadrul programului utilizând adresa furnizată de registrul PC*”.

Excepţie de la această semnificaţie generică s-ar face doar în cazul unei instrucţiuni de salt. De exemplu, instrucţiunea JE (*Jump if Equal*) ar avea următoarea semnificaţie: „*Compară operanzii extraşi de la primele două adrese, dacă sunt egali extrage instrucţiunea de la* ***Adresa3*** *(execută saltul) iar dacă nu, extrage secvenţial următoarea instrucţiune, utilizând adresa furnizată de registrul PC (nu executa saltul deoarece condiţia de salt nu este îndeplinită)*”. Formatul cu trei adrese, cu anumite particularizări, se utilizează de regulă la procesoarele RISC.

Numărul de adrese poate fi în continuare redus dacă se ia decizia ca adresa rezultatului să fie identică cu adresa operandului 2. Rezultă astfel formatul cu două adrese (figura 1.5).

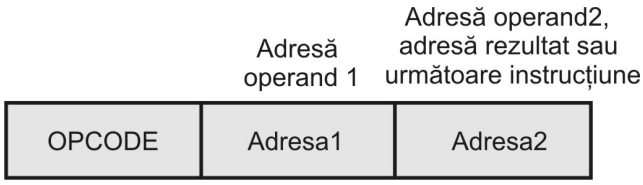


Fig 1.5 Formatul cu două adrese

Semnificaţia generică a unei astfel de instrucţiuni ar fi: „*Extrage operanzii de la primele două adrese, execută operaţia şi depune rezultatul la* ***Adresa2*** *(înlocuind al doilea operand) şi treci la fetch-ul următoarei instrucţiuni din cadrul programului utilizând adresa furnizată de registrul PC*”. Inconvenientul care apare ar fi acela că, prin depunerea rezultatului, operandul 2 se pierde; inconvenientul s-ar face resimţit doar în situaţia în care operandul 2 ar fi necesar în continuare în cadrul programului, pentru operaţii ulterioare.

În cazul formatului cu două adrese, semnificaţia instrucţiunii JE ar fi: „*Testează operandul de la* ***Adresa1****, dacă este zero extrage instrucţiunea de la* ***Adresa2*** *(execută saltul) iar dacă nu, extrage secvenţial următoarea instrucţiune, utilizând adresa furnizată de registrul PC (nu executa saltul deoarece condiţia de salt nu este îndeplinită)*”.

Inconvenientul pierderii operandului 2 (prin depunerea rezultatului), poate fi evitat dacă, în structura procesorului se introduce un registru special numit acumulator (A), care să aibă rolul de a memora rezultatele (rezultatul nu se mai depune la **Adresa2** ci în acumulator). Semnificaţia generică a instrucţiunii cu două adrese devine în acest caz: „*Extrage operanzii de la primele două adrese, execută operaţia şi depune rezultatul în registrul A şi treci la fetch-ul următoarei instrucţiuni din cadrul programului utilizând adresa furnizată de registrul PC*”. Evident că în setul de instrucţiuni aferent unui astfel de procesor, va trebui să existe şi o instrucţiune ST A (*STore A*) care ar putea avea semnificaţia: „*Memorează conţinutul registrului A la* ***Adresa1*** *(câmpul* ***Adresa2*** *nefiind utilizat în cazul acestei instrucţiuni)*”. Instrucţiunea ST A va permite, atunci când se doreşte, „eliberarea” registrului A (transferul rezultatului din acumulator într-o locaţie de memorie).

În cazul utilizării registrului A, semnificaţia instrucţiunii de ramificaţie (JE) ar putea fi: „*Compară operandul de la* ***Adresa1*** *cu conţinutul registrului A, dacă sunt egali extrage instrucţiunea de la* ***Adresa2*** *(execută saltul) iar dacă nu, extrage secvenţial următoarea instrucţiune, utilizând adresa furnizată de registrul PC (nu executa saltul deoarece condiţia de salt nu este îndeplinită)*”. Formatul cu două adrese, cu anumite particularizări, se utilizează de regulă la procesoarele CISC.

În cazul procesoarelor „orientate pe acumulator” (procesoarele care conţin intern un registru acumulator), este posibil şi formatul cu o singură adresă (figura 1.6).

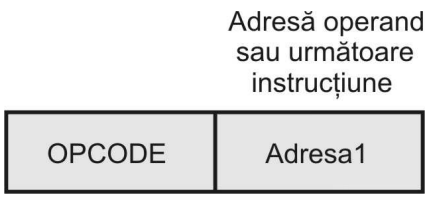


Fig 1.6 Formatul cu o adresă

Semnificaţia generică a unei astfel de instrucţiuni ar fi: „*Extrage primul operand de la* ***Adres1****, al doilea din registrul A, execută operaţia şi depune rezultatul în A (înlocuind al doilea operand) şi treci la fetch-ul următoarei instrucţiuni din cadrul programului utilizând adresa furnizată de registrul PC*”. Evident că şi în acest caz este necesar ca în setul de instrucţiuni al procesorului să existe înstrucţiunea ST A care va permite, atunci când se doreşte, „eliberarea” registrului A (transferul rezultatului din acumulator într-o locaţie de memorie).

În cazul formatului cu o singură adresă, semnificaţia instrucţiunii de ramificaţie (JE) ar putea fi: „*Testează conţinutul registrului A, dacă este zero extrage instrucţiunea de la* ***Adresa1*** *(execută saltul) iar dacă nu, extrage secvenţial următoarea instrucţiune, utilizând adresa furnizată de registrul PC (nu executa saltul deoarece condiţia de salt nu este îndeplinită)*”.

Formatul cu o singură adresă îl mai regăsim astăzi doar la procesoarele foarte simle, orientate pe acumulator şi cu set redus de instrucţiuni (microcontrolere simple).

1.3.2 Formatul instrucţiunii la procesoarele CISC

Procesoarele actuale conţin în structura lor, un număr de registre de lucru (registre interne), numite registre generale. Numărul de registre generale reprezintă de regulă o putere a lui 2. Prin urmare, găsim astăzi procesoare cu 8, 16, 32, 64 sau chiar 128 registre generale. Spre deosebire de procesorul care nu dispune de registre generale, caz în care operanzii instrucţiunii pot fi memoraţi doar în locaţii de memorie, în cazul unui procesor bazat pe registre generale, operanzii instrucţiunii pot fi memoraţi fie în locaţii de memorie fie în registre generale. Prin urmare, în cazul procesorului bazat pe registre generale, o instrucţiune cu doi operanzi ar putea avea ambii operanzi în memorie, ambii operanzi în registre sau un operand în memorie şi celălalt în registru. Această flexibilitate (relativă la localizarea operanzilor) o regăsim implementată la procesoarele CISC. Remintin şi faptul că la procesoarele CISC se utilizează de regulă formatul cu două adrese.

În cazul procesoarelor CISC (orientate desigur pe registre generale), adresa unui operand se specifică (în codul instrucţiunii) cu ajutorul a două câmpuri, **mod de adresare** şi respectiv **registru** (registrul general utilizat pentru localizarea operandului). Pentru a clarifica modul de localizare a operandului, redăm în figura 1.7 formatul cu două adrese pentru un procesor CISC.

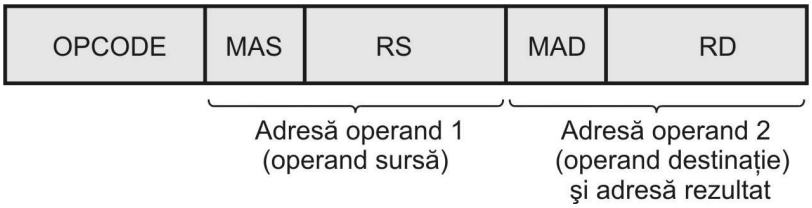


Fig 1.7 Formatul cu două adrese utilizat la procesoarele CISC

Semnificaţia câmpului OPCODE este cunoscută. Celelalte patru câmpuri au următoarele semnificaţii:

MAS –**mod adresare sursă** (mod de adresare operand sursă)
MAD –**mod adresare destinaţie** (mod de adresare operand destinaţie)
RS –**registrul sursă** (registrul general utilizat pentru localizarea operandului sursă)
RD –**registrul destinaţie** (registrul general utilizat pentru localizarea operandului destinaţie)

Să presupunem că procesorul CISC în discuţie are 16 registre generale ($R_0 \div R_{15}$) şi operează cu patru moduri de adresare. În câmpul RS (RD) se codifică de fapt adresa registrului general utilizat pentru localizarea operandului sursă (destinaţie). Având în vedere că procesorul CISC exemplificat conţine 16 registre generale, însemnă că adresa unui registru trebuie codificată pe patru biţi (2^4=16). Prin urmare, câmpurile RS şi respectiv RD vor fi câmpuri de 4 biţi.
În tabelul 1.2 sunt exemplificate patru moduri de adresare (cele mai uzuale).

Mod de adresare	**MAS/MAD** (exemplu de codificare)	**Localizarea operandului**	**Instrucţiuni cu doi operanzi** (exemplificare în limbaj de asamblare)
Imediat	00	Operand de tip constantă	LD R0,4 ;$R_0 \leftarrow 4$ (încarcă constanta 4 în regiustrul R_0)
Direct	01	Operand în registru	ADD R5,R3 ;$R_5 \leftarrow R_5+R_3$ (adună R_3 cu R_5 şi depune rezultatul în R_5)
Indirect	10	Operand în memorie la adresa specificată de registru	MOV R2,(R6) ;$R_2 \leftarrow LocMem|_{R_6}$ (încarcă în R_2 conţinutul locaţiei de memorie adresată de R_6)
Indexat	11	Operand în memorie la adresa dată de registru plus index	AND R4,6(R7) ;$R_4 \leftarrow R_4 \bullet LocMem|_{R_7+6}$ (operaţia logică SI între R_4 şi conţinutul locaţiei de memorie adresată de R_7+6, cu depunerea rezultatului în R_4)

Tabelul 1.2 Moduri de adresare şi exemple de instrucţiuni cu doi operanzi

Pentru a codifica patru moduri de adresare sunt necesari doi biţi. Prin urmare, la procesorul CISC exemplificat, câmpurile MAS şi respectiv MAD sunt câmpuri de doi biţi. Pe prima coloană din tabelul 1.2 sunt redate denumirile celor patru moduri de adresare iar pe coloana a doua (MAS/MAD) se propune un mod de codificare a celor patru moduri de adresare. Pe a treia coloană se explicitează localizarea operandului iar pe ultima coloană sunt redate, în limbaj de asamblare, exemple de instrucţiuni care utilizează diferite moduri de adresare. Pentru aceste exemple am adoptat convenţia (uzuală în limbajul de asamblare): primul operand specificat (în mnemonica instrucţiunii) este operandul destinaţie iar cel de-al doilea este operandul sursă.

În cazul instrucţiunii LD R0,4 operandul sursă este constanta 4, ceea ce implică MAS=00 (mod de adresare **imediat** la sursă) iar operandul destinaţie esta registrul R_0, ceea ce implică MAD=01 (mod de adresare **direct** la destinaţie).

În cazul instrucţiunii ADD R5,R3 operandul sursă este registrul R_3, ceea ce implică MAS=01 iar operandul destinaţie esta registrul R_5, ceea ce implică MAD=01 (mod de adresare **direct** atât la sursă cât şi la destinaţie).

În cazul instrucţiunii MOV R2,(R6) operandul sursă este conţinutul locaţiei de memorie adresată de registrul R_6 ($LocMem|_{R_6}$), ceea ce implică MAS=10 (mod de adresare **indirect** la sursă) iar operandul destinaţie este registrul R_2 ceea ce implică MAD=01 (mod de adresare **direct** la destinaţie).

În sfârşit, în cazul instrucţiunii AND R4,6(R7) operandul sursă este conţinutul locaţiei de memorie selectată de adresa obţinută prin adunarea indexului (care are valoarea 6 în acest exemplu) la conţinutul registrului R_7 ($LocMem|_{R_7+6}$), ceea ce implică MAS=11 (mod de adresare **indexat** la sursă) iar operandul destinaţie esta registrul R_4, ceea ce implică MAD=01 (mod de adresare **direct** la destinaţie).

Să notăm că modul de adresare imediat poate fi utilizat doar la operandul sursă; o constantă poate fi specificată ca operand sursă dar nu poate fi specificată ca operand destinaţie. Prin urmare, un cod de instrucţiune cu mod de adresare imediat la destinaţie va fi cod ilegal (instrucţiune ilegală).

Să presupunem că proiectantul a rezervat patru biţi pentru câmpul OPCODE şi că a codificat instrucţiunile (cu ajutorul câmpului OPCODE) ca în tabelul 1.3.

OPCODE	Instrucţiune
0000	LD *dest*, *src*
0001	ADD *dest*,*src*
0010	AND *dest*,*src*
0011	MOV *dest*, *src*
.....	.....
.....	.....

Tabelul 1.3 Exemplu de codificarea a instrucţiunilor

Ţinând cont că MAS şi respectiv MAD sunt câmpuri de doi biţi (procesorul CISC exemplificat operează cu patru moduri de adresare), câmpurile RS şi respectiv RD sunt câmpuri de patru biţi (procesorul CISC exemplificat are 16 registre generale) şi că formatul de instrucţiune utilizat este cel cu două adrese (figura 1.7), rezultă că instrucţiunile acestui procesor CISC vor fi codificate pe 16 biţi (OPCODE+MAS+RS+MAD+RD= =4+2+4+2+4=16 biţi). În aceste condiţii, codul maşină (codul binar) aferent instrucţiunii MOV R2,(R6) ar fi: 0011100110010010_2 (OPCODE=0011, MAS=10, RS=0110, MAD=01, RD=0010).

Codul maşină aferent instrucţiunii LD R0,4 ar fi: $000000XXXX010000_2$ (OPCODE=0000, MAS=00, RS=XXXX, MAD=01, RD=0000). Câmpul RS este neutilizat în acest caz deoarece operandul sursă este o constantă (mod de adresare imediat la sursă). Prin urmare, cei patru biţi din componenţa câmpului RS pot lua orice valoare (RS=XXXX). Pe de altă parte, constanta 4 (operandul sursă) nu a fost codificată în cei 16 biţi ce compun codul instrucţiunii. O soluţie ar fi ca, pentru această instrucţiune, să recurgem la extensia codului de la 16 la 32 de biţi. În primul cuvânt de 16 biţi se codifică

codul propriu-zis al instrucţiunii ($000000XXXX010000_2$) iar în al doilea cuvânt de 16 biţi se codifică constanta 4 (0000000000000100_2). Prin urmare, codul maşină aferent instrucţiunii LD R0,4 va ocupa două locaţii succesive în memorie (figura 1.8). Iată cum, la CISC, se ajunge la formatul de lungime variabilă !

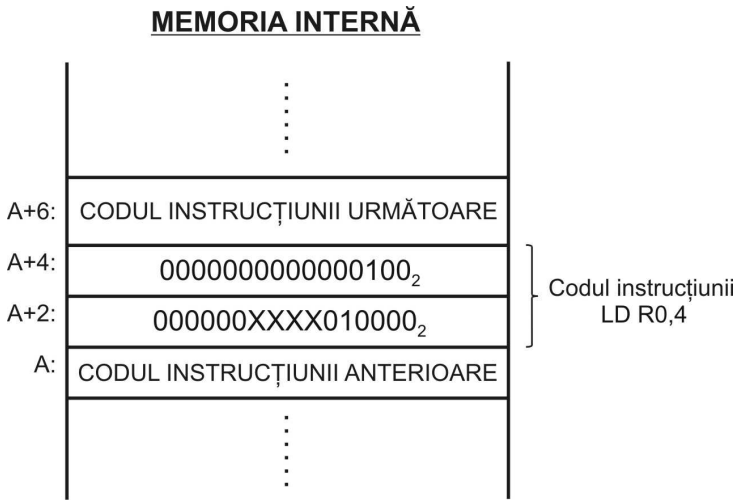


Fig 1.8 Codificarea instrucţiunii LD R0,4 în memorie

În mod similar se poate codifica instrucţiunea AND R4, 6(R7) unde avem de-a face din nou cu o constantă (constanta 6). Chiar dacă constanta va fi utilizată ca index şi nu ca operand imediat, din punctul de vedere al codificării binare, tot o constantă rămâne. Adresăm cititorului invitaţia de a face şi alte exerciţii de codificare.

Formatul din figura 1.7 îl vom regăsi la multe procesoare CISC, acest format fiind valabil pentru instrucţiunile cu doi operanzi. În setul de instrucţiuni aferent procesorului există şi alte tipuri de instrucţiuni, cum ar fi: instrucţiuni cu un singur operand, instrucţiuni fără operanzi, etc. Pentru instrucţiunile cu un singur operand, în formatul instrucţiunii se va codifica o singură adresă (adresa operandului destinaţie). Acest format este redat în figura 1.9.

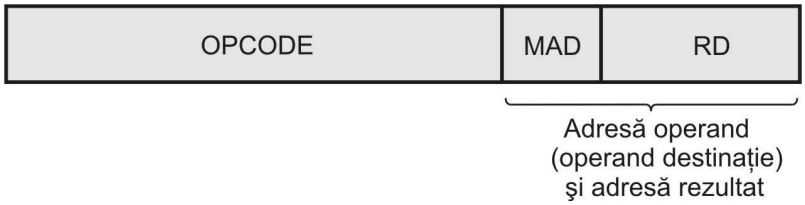


Fig 1.9 Formatul instrucţiunilor cu un singur operand, utilizat la procesoarele CISC

Formatul instrucţiunilor cu un singur operand (figura 1.9) se obţine operând o simplificare în formatul instructiunilor cu doi operanzi (figura 1.7): se elimină câmpurile MAS şi RS (se elimină adresa operandului sursă). Câmpurile MAD şi RD vor specifica nu numai adresa unicului operand specificat de instrucţiune (operandul destinaţie) ci şi adresa rezultatului. În tabelul 1.4 sunt exemplificate câteva instrucţiuni cu un singur operand.

Mod de adresare	MAD	Localizarea operandului	Instrucţiuni cu un singur operand (exemplificare în limbajul de asamblare)
Direct	01	Operand în registru	NEG R5 ; $R_5 \leftarrow \overline{R_5}$ (negare bit cu bit a conţinutului registrului R_5)
Indirect	10	Operand în memorie la adresa specificată de registru	DEC (R6) ; $LocMem\|_{R_6} \leftarrow LocMem\|_{R_6} - 1$ (decrementarea conţinutului locaţiei de memorie de la adresa dată de registrul R_6)
Indexat	11	Operand în memorie la adresa specificată de registru plus un index	INC 6(R7) ; $LocMem\|_{R_7+6} \leftarrow LocMem\|_{R_7+6} + 1$ (incrementarea conţinutului locaţiei de memorie de la adresa R_7+6)

Tabelul 1.4 Moduri de adresare şi exemple de instrucţiuni cu un singur operand

Instrucţunile exemplificate în tabelul 1.4 operează cu trei moduri de adresare: direct, indirect şi indexat. Precizarea pe care am făcut-o la instrucţiunile cu doi operanzi rămâne valabilă şi pentru instrucţiunile cu un singur operand: modul de adresare imediat nu poate fi utilizat la operandul destinaţie deoarece o constantă nu poate fi specificată ca operand destinaţie. Prin urmare, un cod de instrucţiune cu mod de adresare imediat la destinaţie va fi considerat cod ilegal (instrucţiune ilegală).

Pentru instrucţiunile fără operanzi codul instrucţiunii se simplifică şi mai mult deoarece, în codul unei asemenea instrucţiuni, nu mai trebuie specificată nici-o adresă. Prin urmare, formatul instrucţiunii fără operanzi va conţine doar câmpul OPCODE (figura 1.10).

Fig 1.10 Formatul instrucţiunilor fără operanzi, utilizat la procesoarele CISC

Instrucţiunile de control, de exemplu, sunt instrucţiuni fără operanzi. Astfel de instrucţiuni regăsim la toate procesoarele:

HALT ;oprire. Această intrucţiune blochează tactul procesorului.

EI ;validare întreruperi (*Enable Interrupts*). Această instrucţiune ;validează întreruperile externe.

DI ;invalidare întreruperi (*Disable Interrupts*). Această instrucţiune ;invalidează întreruperile externe.

NOP ;nici o operaţie (*No OPeration*). Această instrucţiune care nu face ;nici-o operaţie poate fi utilizată la proiectarea rutinelor de ;întârziere (*delay*). Este utilă de asemenea compilatoarelor cu ;reorganizare care, dupa translatarea programului sursă în cod ;obiect, parcurg o a doua fază de optimizare a codului obiect, ;reordonând instrucţiunile din cadrul acestuia. Scopul acestei ;reordonări, care se face astfel încât să nu afecteze logica ;programului, este acela de a minimiza timpul global de execuţie ;a aplicaţiei. Optimizarea vizează desigur procesoarele moderne ;care operează cu suprapunere de faze (paralelism la nivelul ;instrucţiunii).

Observaţie:
Cele trei formate de instrucţiune prezentate în figurile 1.7, 1.9 şi 1.10 sunt principalele formate utilizate la procesoarele CISC. Aceste formate acoperă trei clase importante de instrucţiuni: instrucţiunile cu doi operanzi, instrucţiunile cu un operand şi instrucţiunile fără operanzi. În setul de instrucţiuni există şi instrucţiuni mai speciale cum ar fi instrucţiunile de salt. Instrucţiunile de salt sunt de două tipuri: salturi directe şi respectiv salturi indirecte. Salturile directe fac saltul la o adresă codificată explicit în codul instrucţiunii. Codul maşină aferent unei asemenea instrucţiuni conţine deci o constantă, interpretată ca adresă de salt. Se poate prin urmare recurge la aceeaşi extensie a codului pe două cuvinte succesive (ca în figura 1.8); primul cuvânt va conţine codul propriu-zis iar al doilea va conţine adresa de salt. Salturile indirecte fac salt la o adresă conţinută într-un anumit registru (indirectare prin registru). Aceste instrucţiuni folosesc prin urmare modul de adresare indirect. Pentru codificarea salturilor indirecte s-ar putea deci recurge la formatul din figura 1.9 în cadrul căruia s-ar accepta doar modurile de adresare care conduc efectiv la indirectare: modul indirect şi respectiv indexat.

1.3.3 Formatul instrucţiunii la procesoarele RISC

Procesoarele RISC se subordonează conceptului „set minimal de instrucţiuni”. Prin urmare, setul de instrucţiuni aferent procesorului RISC conţine foarte puţine instrucţiuni şi toate instrucţiunile sunt simple. Cu alte cuvinte, setul RISC conţine doar instrucţiuni indispensabile compilatorului (fără de care compilatorul nu poate rezolva translatarea aplicaţiei, din limbajul de nivel înalt în cod maşină). Pe lângă conceptul setului minimal, evitarea instrucţiunilor complexe este un alt scop important al arhitecturii RISC. Consecinţa imediată este evident simplificarea structurii unităţii de control. Spre deosebire de unitatea de control aferentă procesorului CISC care are o structură complexă şi care este de regulă microprogramată, unitatea de control aferentă procesorului RISC are o structură simplă şi se implementează de regulă în variantă cablată.

Dat fiind faptul că, în arhitectura RISC, singurele instrucţiuni care pot specifica operanzi în memorie sunt instrucţunile ***load*** şi respectiv ***store***, procesorul RISC se mai numeşte şi „maşină *load-store*”. Toate celelalte instrucţiuni pot specifica operanzi locaţi exclusiv în registre. Un operand aflat într-o locaţie de memorie poate fi încărcat într-un registru general cu ajutorul instrucţiunii ***load***. Instrucţiunea ***store*** realizează operaţia

inversă: transferă (memorează) conţinutul unui registru general într-o locaţie de memorie. Prin urmare, cu ajutorul instrucţiunilor ***load*** şi ***store*** se pot efectua transferurile (de date) procesor-memorie. Celelalte instrucţiuni (instrucţiunile *non-load-store*) operează strict pe registre (preiau operanzii din registre şi depun rezultatul în registru). Cu alte cuvinte, instrucţiunile *non-load-store* operează cu un singur mod de adresare: modul direct. Din acest motiv câmpurile MAS şi respectiv MAD (câmpurile **mod de adresare**) nu mai sunt necesare şi vor fi eliminate din formatul instrucţiunii RISC.

Biţii economisiţi prin eliminarea câmpurilor MAS şi respectiv MAD pot fi alocaţi celei de-a treia adrese de registru. Iată de ce la RISC se utilizează de regulă formatul cu trei adrese (figura 1.11). Pentru că în formatul redat în figura 1.11 sunt codificate trei adrese de registru, acest format este denumit adesea **formatul R-R-R** (formatul registru-registru-registru).

Cea de-a treia adresă conferă o flexibilitate suplimentară relativă la localizarea celor doi operanzi şi respectiv rezultatului (operanzii sunt preluaţi din două registre iar rezultatul va fi depus într-un al treilea registru). Cu alte cuvinte, depunerea rezultatului nu mai conduce inevitabil la pierderea unuia dintre operanzi.

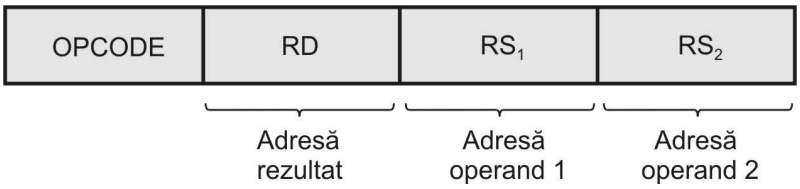


Fig 1.11 Formatul R-R-R, utilizat la procesoarele RISC

Setul de instrucţiuni maşină trebuie să confere programatorului (compilatorului) şi posibilitatea de a specifica operanzi de tip constantă (operanzi imediaţi). Această cerinţă nu ţine cont de tipul procesorului, RISC sau CISC. Prin urmare şi la RISC trebuie să existe instrucţiuni în care unul dintre cei doi operanzi (de exemplu operandul sursă 2) să fie o constantă şi nu conţinutul unui registru. Această cerinţă conduce la cel de-al doilea format uzual la procesoarele RISC: **formatul R-R-I** (formatul registru-registru-imediat sau formatul registru-registru-constantă). Acest format este redat în figura 1.12.

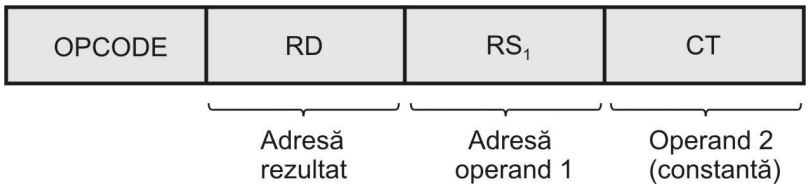


Fig 1.12 Formatul R-R-I, utilizat la procesoarele RISC

Fiecare instrucţiune din clasa R-R-R are o instrucţiune corespondentă în clasa R-R-I. De exemplu, corespondenta instrucţiunii ADD R_i,R_j,R_k ($R_i \leftarrow R_j+R_k$) din clasa R-R-R, este

instrucţiunea ADD R_i,R_j,CT ($R_i \leftarrow R_j$+CT) din clasa R-R-I. Notaţia CT semnifică operand de tip constantă, precizat prin valoare. În tabelul 1.5 sunt redate exemple de instrucţiuni din clasele R-R-R şi respectiv R-R-I.

Clasa	Instrucţiune	Operaţie efectuată
R-R-R	ADD R1, R7,R5	$R_1 \leftarrow R_7+R_5$ (adunare)
	AND R6, R0,R4	$R_6 \leftarrow R_0 \bullet R_4$ (SI logic)
	XOR R3,R9,R2	$R_3 \leftarrow R_9 \oplus R_2$ (SAU exclusiv)
R-R-I	ADD R1, R7,24	$R_1 \leftarrow R_7+24$ (adunare)
	AND R6, R0,8	$R_6 \leftarrow R_0 \bullet 8$ (SI logic)
	XOR R3,R9,152	$R_3 \leftarrow R_9 \oplus 152$ (SAU exclusiv)

Tabelul 1.5 Exemple de instrucţiuni RISC din clasele R-R-R şi R-R-I

În setul de instrucţiuni aferent procesorului RISC există şi alte tipuri de instrucţiuni: instrucţiuni cu un singur operand, instrucţiuni fără operanzi, instrucţiuni *load-store*, instrucţiuni de salt, etc. Formatele aferente acestor tipuri de instrucţiuni se pot obţine pornind de la formatele R-R-R sau R-R-I, după caz. Pentru a obţine aceste formate, în principiu, trebuie operate simplificări în formatele R-R-R şi respectiv R-R-I în sensul că anumite câmpuri rămân fără semnificaţie şi prin urmare nu vor fi utilizate (nu vor avea relevanţă în faza de execuţie a instrucţiunii).

Pentru instrucţiunile cu un singur operand rămâne valabil formatul R-R-R doar că adresa RS_2 devine nenecesară (nu va fi utilizată) iar la instrucţiunile fără operanzi va fi utilizat doar câmpul OPCODE (toate cele trei adrese devin nenecesare).

Instrucţiunile *load* şi *store* controlează transferurile procesor-memorie (tabelul 1.6). Locaţia de memorie implicată în transfer reprezintă operandul sursă la instrucţiunea *load* (*load* din memorie în registru) şi respectiv operandul destinaţie la instrucţiunea *store* (*store* registru în memorie). Pe de altă parte, un operand aflat în memorie poate fi adresat fie indirect, fie indexat (vezi exemplele prezentate în tabelul 1.6).

Analizând instrucţiunile exemplificate în tabelul 1.6, constatăm că în codul unei instrucţiuni *load* (*store*), pe lângă OPCODE, trebuie codificate următoarele informaţii:

- adresele a două registre generale: registrul de date şi respectiv registrul de adresă (registrul utilizat în transferul de date şi respectiv registrul utilizat pentru adresarea locaţiei de memorie).

- o constantă care va fi utilizată ca index, dacă pentru adresarea locaţiei de memorie se utilizează modul indexat; dacă se utilizează modul indirect constanta devine zero.

O soluţie posibilă pentru codificarea instrucţiunilor *load* şi respectiv *store* este redată în figura 1.13.

Mod de adresare	Instrucţiune	Operaţie efectuată
indirect	LD R2,(R6)	$R_2 \leftarrow LocMem\|_{R_6}$ (încarcă în R_2 conţinutul locaţiei de memorie adresată de R_6)
	ST (R3),R1	$LocMem\|_{R_3} \leftarrow R_1$ (memorează registrul R_1 în locaţia de memorie adresată de R_3)
indexat	LD R7,16(R5)	$R_7 \leftarrow LocMem\|_{R_5+16}$ (încarcă în R_7 conţinutul locaţiei de memorie de la adresa R_5+16)
	ST 124(R4),R0	$LocMem\|_{R_4+124} \leftarrow R_0$ (memorează registrul R_0 în locaţia de memorie de la adresa R_4+124)

Tabelul 1.6 Exemple de instrucţiuni *load* şi *store*

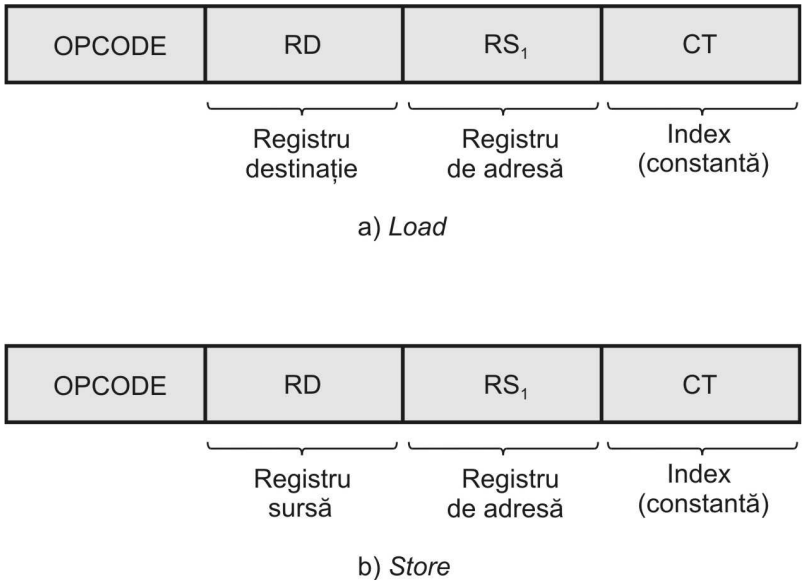


Fig 1.13 Formatul *load*/*store*, utilizat la procesoarele RISC

Dacă comparăm formatul *„load”* cu formatul *„store”* (figura 1.13) şi mai departe, cu formatul R-R-I din figura 1.12 constatăm că, din punct de vedere structural, sunt identice (câmpurile din structura celor trei formate sunt identice; diferenţe apar doar la nivelul semnificaţiei anumitor câmpuri). Prin urmare, pentru instrucţiunile *load* şi respectiv *store* se utiliză de fapt formatul R-R-I din figura 1.12.

Pentru un procesor RISC pe 32 de biţi, dotat cu 32 registre generale, formatele R-R-R şi R-R-I ar putea fi particularizate ca în figura 1.14.

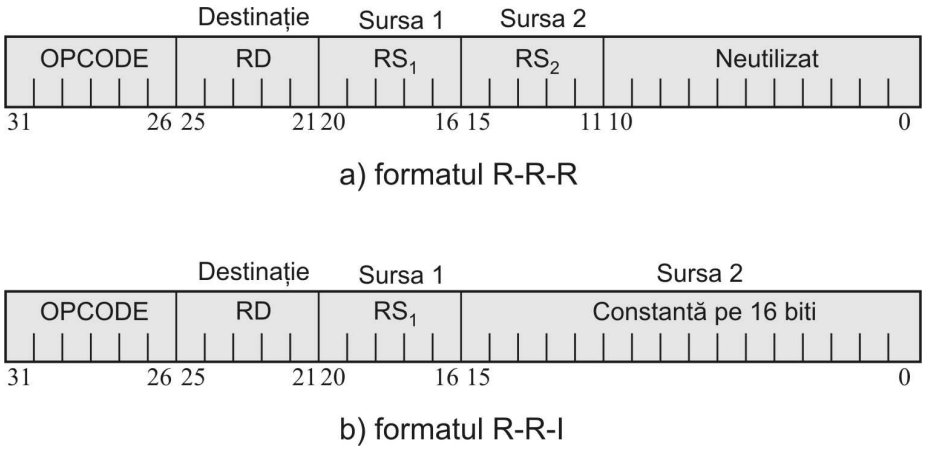


Fig 1.14 Formatele R-R-R şi R-R-I particularizate pentru un procesor RISC pe 32 biţi, dotat cu 32 registre generale

Dacă procesorul are 32 registre generale atunci adresa unui registru trebuie codificată pe 5 biţi (2^5=32). Prin urmare, câmpurile RD, RS_1 şi RS_2 vor fi câmpuri de 5 biţi. Împreună, cele trei câmpuri vor ocupa 15 biţi şi dacă alăturăm şi câmpul OPCODE, devine evident faptul că instrucţiunea nu poate fi codificată pe 16 biţi. Vom recurge deci la un format pe 32 de biţi care se potriveşte de altfel şi cu dimensiunea registrelor şi respectiv locaţiilor de memorie (registre şi respectiv locaţii de memorie pe 32 biţi deoarece procesorul este pe 32 de biţi). Pe de altă parte, în formatul R-R-I trebuie să codificăm o constantă (operandul imediat). Cum tot formatul are lungimea de 32 de biţi, devine evident că nu putem codifica o constantă de aceeaşi lungime. Următoarea variantă de codificare ce trebuie încercată ar fi cea care utilizează constante pe 16 biţi. Această soluţie prezintă anumite avantaje deoarece din două constante pe 16 biţi, prin concatenare, se poate obţine uşor o constantă pe 32 de biţi. Dacă adoptăm această soluţie, câmpurile Destinaţie, Sursa 1 şi Sursa 2 din formatul R-R-I (figura 1.14) însumează 5+5+16=26 biţi. Prin urmare, pentru câmpul OPCODE rămân disponibili 32-26=6 biţi. Un OPCODE pe 6 biţi permite codificarea a 64 de instrucţiuni ceea ce s-ar putea să fie suficient deoarece arhitectura RISC presupune set minimal de instrucţiuni.

Trecerea de la formatul R-R-I la formatul R-R-R (figura 1.14), presupune înlocuirea constantei (codificată pe 16 biţi) cu câmpul RS_2 (codificat pe 5 biţi). Prin urmare, în formatul R-R-R vor rămâne 11 biţi (cei mai puţin semnificativi) neutilizaţi. De altfel, aceşti biţi reprezintă o rezervă pentru eventuala extensie a setului de instrucţiuni. Astfel, pentru

instrucţiunile din clasa R-R-R, biţii 31÷26 reprezintă zona OPCODE iar biţii 10÷0 ar putea reprezenta o zonă de SUBOPCODE. O combinaţie din câmpul OPCODE ar putea specifica o clasă de instrucţiuni (de exemplu, instrucţiuni aritmetice) iar diversele combinaţii din SUBOPCODE ar putea specifica diverse instrucţiuni în cadrul clasei respective. În această situaţie, câmpul SUBOPCODE ar fi aplicat direct pe intrările de control aferente unităţii ALU pentru a selecta funcţia (operaţia) ce va fi efectuată.

În paragrafele 1.3.2 şi 1.3.3 am realizat o scurtă introducere în problematica formatului instrucţiunii utilizat la procesoarele CISC şi respectiv RISC. Definirea setului de instrucţiuni aferent unui procesor reprezintă o problemă de proiectare şi în cadrul acestei probleme trebuie stabilite formatele aferente tuturor tipurilor (claselor) de instrucţiuni din cadrul setului vizat. Proiectarea şi optimizarea setului de instrucţiuni reprezintă prin urmare o problemă de proiectare complexă; în paragrafele 1.3.2 şi 1.3.3 am realizat doar o succintă incursiune în „filozofiile” CISC şi respectiv RISC (din perspectiva formatului instrucţiunii).

1.4 Ierarhizarea memoriei

Toate informaţiile (programe şi respectiv date) procesate de către calculator sunt stocate în memorie. Orice aplicaţie soft (program) produce rezultate (date) care vor fi stocate tot în memorie de unde, prin intermediul unor echipamente periferice de ieşire, vor putea fi livrate utilizatorului.

Software-ul procesat de calculator se împarte în două clase mari:

- soft de bază
- soft de aplicaţii

În clasa soft de aplicaţii regăsim toate programele dedicate rezolvării (automate) de probleme, din cele mai diverse domenii de activitate. Există aplicaţii utilizate în domeniul ştiinţific care permit cercetătorilor să facă investigaţii şi experimente ştiinţifice extrem de complexe, există aplicaţii utilizate în domeniul economic care permit gestiunea eficientă a resurselor unei companii, există aplicaţii utilizate în domeniul ingineriei prin care se eficientizează activitatea de proiectare, există aplicaţii utilizate în domeniul financiar care permit gestiunea resurselor financiare ale unei bănci, există multiple aplicaţii de uz general şi multiple aplicaţii dedicate; toate acestea conduc la inconfortabila concluzie că societatea de astăzi este puternic dependentă de calculator.

În clasa soft de bază regăsim sistemele de operare, compilatoarele, mediile de programare, etc. Softul de bază face posibilă procesarea programelor de aplicatii. Mediile de programare permit programatorilor sa scrie programe de aplicatii. Compilatoarele translateaza în cod maşină (format executabil), programele de aplicatii (programele sursă) scrise de către programatori în limbaje de nivel înalt. Sistemul de operare (SO) gestionează toate resursele *hardware* ale calculatorului. Pentru a putea fi executat, SO alocă fiecărui program de aplicaţie două resurse importante: **timp procesor** şi **spaţiu de memorie**. SO gestionează de asemenea şi **operaţiile de intrare-ieşire** (operaţiile de transfer cu perifericele). SO este în fond un pachet de programe şi fiecare periferic are, în cadrul acestui pachet, un program special numit *driver* (*driver* de disc, *driver* de imprimantă, *driver* de tastatură, *driver* de *display*, etc). *Driver*-ul asociat unui periferic controlează toate transferurile de date pe care calculatorul le poate iniţia pe perifericul respectiv. Programele de aplicaţii pot iniţia operaţii de intrare-ieşire (operaţii de transfer cu perifericele) prin intermediul unor instrucţiuni dedicate. În realitate, aceste instrucţiuni iniţiază nişte

întreruperi soft prin care se lansează în executie *driver*-ele de periferice. În această manieră, se suspendă temporar un program de aplicaţie şi se lansează în execuţie un *handler* (din componenţa sistemului de operare!). *Handler*-ul va executa operaţia de intrare-ieşire solicitată de către programul de aplicaţie (operaţia de transfer cu perifericul) şi se va încheia cu o revenire în programul de aplicaţie anterior suspendat. Prin urmare, după efectuarea operaţiei de intrare-ieşire programul de aplicaţie va fi reluat (exact din locul din care a fost suspendat) şi va fi continuat.

Privită de la nivelul sistemului de operare, memoria unui calculator are o structură ierarhizată pe trei nivele:

- **Nivelul 1** este reprezentat de registrele generale implementate în interiorul procesorului; este prin urmare o memorie de foarte mică capacitate (8, 16, 32 sau 64 de registre) dar de viteză foarte mare (accesul la un registru implementat în interiorul procesorului este foarte rapid).

- **Nivelul 2** este reprezentat de memoria internă, denumită uneori şi memorie principală sau memorie centrală. Memoria internă este implementată cu circuite speciale de memorie RAM (*Random Access Memory*), cele mai utilizate fiind circuitele RAM-dinamic. Capacitatea memoriei principale este relativ mare dar, comparativ cu nivelul 1, viteza este mult mai mică (timpul de acces la o locatie de memorie principală este semnificativ mai mare decât timpul de acces la un registru).

- **Nivelul 3** este reprezentat de memoria externă denumită şi memorie secundară. Acest nivel este implementat fizic sub forma unor echipamente periferice de memorare, cel mai important fiind discul. Memoria externă are o capacitate semnificativ mai mare decât memoria internă (aproximativ de $10^2 \div 10^3$ ori mai mare) dar şi o viteză semnificativ mai mică (aproximativ de $10^4 \div 10^5$ ori mai mică).

Spre deosebire de nivelele 1 şi respectiv 2 care sunt volatile, memoria externă are marele avantaj că este nevolatilă (nu-şi pierde conţinutul în momentul decuplării calculatorului de la sursa de energie electrică). Este motivul pentru care toate programele şi respectiv datele sunt stocate în memoria externă (pe disc). Procesorul nu poate însă accesa direct informaţii în memoria externă. Prin urmare, dacă se doreşte lansarea în execuţie a unui program, acesta trebuie în prealabil transferat din memoria externă în memoria internă (de pe disc în RAM). Aceste transferuri operează pe blocuri mari de informaţie numite fişiere şi se fac în regim DMA (*Direct Memory Access*). Unitatea de informaţie transferabilă între RAM şi disc este fişierul deoarece informaţia stocată în memoria externă (pe disc) este organizată în fişiere, fişierele fiind de două tipuri: de cod şi respectiv de date.

Locaţiile de memorie internă (precum şi registrele generale) pot fi accesate direct de către procesor. O locaţie de memorie internă va fi accesată de procesor fie pentru a citi informaţia memorată în locaţia respectivă, fie pentru a scrie o nouă informaţie în locaţia respectivă. Unitatea adresabilă în memoria internă este octetul (*byte*-ul). Prin urmare unitatea minimă transferabilă între memoria internă şi procesor este octetul. Dacă, de exemplu, procesorul operează pe 32 de biţi atunci şi locaţiile memoriei interne vor fi pe 32 de biţi. Prin urmare, printr-un singur acces la memorie procesorul va putea citi sau scrie un cuvânt (o locaţie) de 32 de biţi. În setul de instrucţiuni al procesorului vor exista

instrucţiuni care operează pe cuvânt (32 de biţi) dar şi instrucţiuni care operează pe octet (8 biţi).

Prin accese succesive la memoria internă vor fi extrase din memorie instrucţiunile unui program, în timpul execuţiei acestuia. După *fetch*-ul şi execuţia instrucţiunii curente se trece la *fetch*-ul şi execuţia instrucţiunii următoare.

1.5 Registrele generale

Aşa cum s-a arătat la 1.3.2 şi 1.3.3, registrele generale sunt implementate în procesor şi reprezintă memoria de pe nivelul 1. În faza de compilare a unui program, pe lângă translatarea acestuia din limbajul de nivel înalt în cod maşină, compilatorul va face şi alocarea memoriei pentru variabilele definite în program. Pentru a optimiza alocarea memoriei, compilatorul analizează frecvenţa utilizării diverselor variabile pe parcursul execuţiei programului. Variabilele cele mai frecvent utilizate vor fi alocate în registre (nivelul 1). După epuizarea registrelor se vor face alocări în locaţii de memorie (nivelul 2). Ţinând cont de faptul că accesul la un registru este mult mai rapid decât accesul la o locaţie de memorie, prin acest sistem de alocare se urmăreşte minimizarea timpului de procesare a programului.

Procesoarele CISC sunt de regulă dotate cu un set mai redus de registre generale (8, 16 sau 32 registre) în timp ce procesoarele RISC sunt de regulă dotate cu un set mai extins de registre (32, 64 sau chiar 128). Lungimea registrului general corespunde structurii procesorului; un procesor pe 32 de biţi va avea registre pe 32 de biţi, ALU pe 32 de biţi, busuri interne pe 32 de biţi, etc.

Un registru pe n biţi este format din n bistabili asociaţi. Se utilizează bistabili D sau bistabili J-K. Bistabilii D au avantajul că necesită o singură linie de intrare date per bit memorat (intrarea D), în timp ce bistabilii J-K revendică două linii de intrare per bit memorat (intrările J şi K).

Un registru pe 8 biţi implementat cu bistabili D este prezentat în figura 1.15.

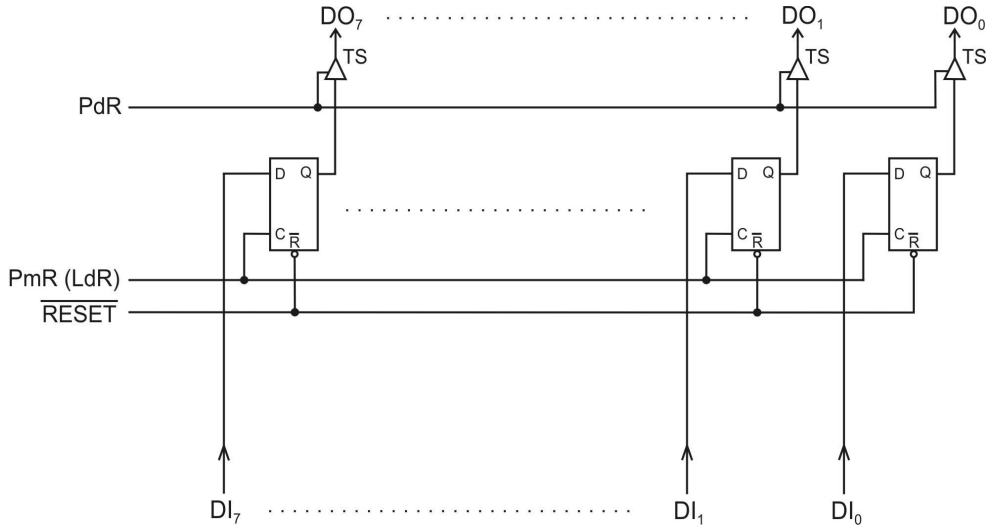


Fig 1.15 Registru pe 8 biţi implementat cu bistabili D

Semnificaţia notaţiilor utilizate în figura 1.15 este următoarea:

DI_{7-0} -liniile de intrare date în registrul R
DO_{7-0} -liniile de ieşire date din registrul R
$\overline{RESET}$ -comanda de ştergere registru, activă pe ”0” (R←0)
LdR (PmR) -„*load register*” („primeşte registru”); comanda de încărcare (scriere) a datelor de intrare DI_{7-0} în registrul R.
PdR -„predă registru”; comanda de citire a conţinutului registrului R pe ieşirile de date DO_{7-0}. Această comandă emite conţinutul registrului pe liniile de ieşire date DO_{7-0}.

Prin urmare, două operaţii sunt posibile în registrul R:

- **Scriere**: datele de inrare DI_{7-0} sunt încărcate (memorate) în R prin activarea comenzii LdR (PmR).

- **Citire**: conţinutul registrului R este citit (emis) pe ieşirile de date DO_{7-0} prin activarea comenzii PdR.

Dacă 2^n astfel de registre se integrează într-o structură (plan) de memorie cu 2^n locaţii pe 8 biţi şi dacă acestui plan de memorie statică rapidă (*register file*) i se asociază un decodificator de adrese cu n intrări şi 2^n ieşiri, fiecare ieşre selectând o locaţie din plan (un registru), se obţine un set de 2^n registre generale pe 8 biţi; vezi structura din figura 1.16 (elaborată pentru cazul particular n=3 şi respectiv registre pe 32 de biţi şi nu pe 8 biţi), structură ce va fi explicată în cele ce urmează).

1.5.1 Registre generale implementate în structuri uniport

Considerăm un set de 8 registre generale pe 32 de biţi (particularizare). Acest set poate fi implementat într-o structură de memorie (statică rapidă) uniport de forma celei din figura 1.16. Semnificaţia notaţiilor utilizate în figura 1.16 este următoarea:

A_{2-0} -intrările de adresă; selectează registrul general
DEC 3:8 -decodificator de adrese
DI_{31-0} -intrările de date
DO_{31-0} -ieşirile de date
WRITE (PmRG) -comanda de scriere („primeşte RG”); cu un impuls în ”1” pe intrarea WRITE se face scriere în registrul general selectat de adresa prezentă pe intrările A_{2-0}.
SEL_i -linia de selecţie a locaţiei i din planul de memorie M (linia de selecţie a registrului R_i).
M -plan de memorie statică rapidă cu dimensiunea de 8 locaţii a câte 32 biţi. Fiecare locaţie reprezintă un registru general. Aceast plan de memorie

implementat cu bistabili D funcţionează conform schemei de principiu din figura 1.17.

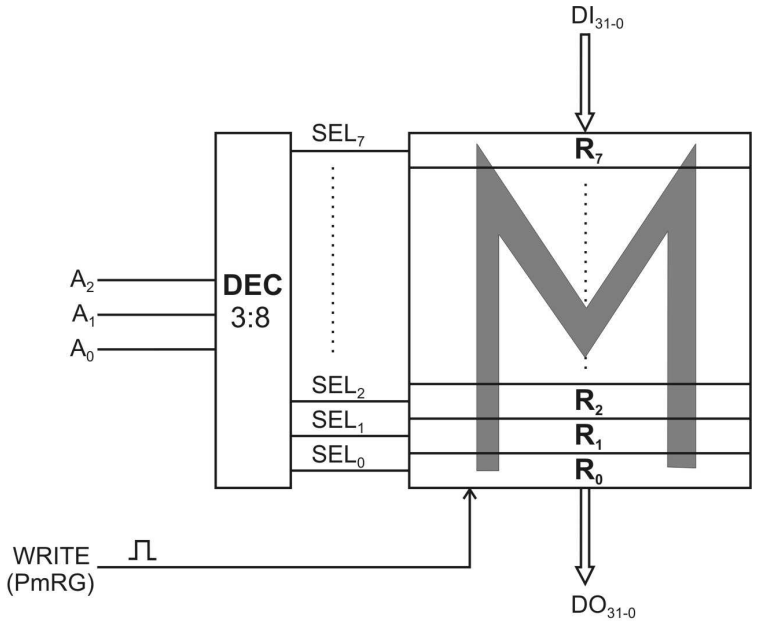


Fig 1.16 Structura uniport aferentă unui set de 8 registre generale pe 32 biţi

Pentru simplificare, în figura 1.17 sunt reprezentate doar locaţiile i-1, i şi respectiv i+1 din planul M (doar trei registre: R_{i-1}, R_i şi R_{i+1}), selectate de liniile SEL_{i-1}, SEL_i, şi respectiv SEL_{i+1}. În total există 8 astfel de locaţii în planul M (8 registre), fiecare fiind compusă din 32 de bistabili asociaţi (registre pe 32 biţi).
În planul M din figura 1.16 (detaliat în figura 1.17) sunt posibile două operaţii:

- **Citirea conţinutului registrului R_i:**
  Adresa de valoare i aplicată pe intrările de adresă A_{2-0} va activa (în ”1”) doar linia de selecţie SEL_i; celelalte linii de selecţie rămân inactive (în ”0”). Cu SEL_i=1 se validează întreg setul de 32 porţi SI (notate cu * în figura 1.17) situate pe ieşirile Q aferente celor 32 de bistabili ce compun locaţia i (registrul R_i). Conţinutul registrului R_i se va propaga prin cele 32 porţi SI menţionate şi mai departe prin cele 32 de porţi SAU (care reunesc printr-o funcţie SAU ieşirile tuturor celor 8 registre), devenind disponibil pe ieşirile de date DO_{31-0}.

- **Scrierea unor noi informaţii în registrul R_i**:
  Ca la citire, adresa de valoare i aplicată pe intrările de adresă A_{2-0} va activa (în ”1”) doar linia de selecţie SEL_i; celelalte linii de selecţie rămân inactive (în ”0”). Suplimentar, data de scris în registrul R_i (un cuvânt de 32 biţi) se aplică pe intrările de date DI_{31-0}. Având cei 32 de biţi de date ce urmează a fi scrişi în registrul R_i, prezenţi şi stabili pe intrările DI_{31-0}, se activează comanda WRITE (PmRG).

  Prin urmare, selecţia locaţiei i (registrului R_i) se face, ca şi în cazul citirii, cu SEL_i=1. Cu SEL_i=1 înseamnă că doar poarta SI notată cu ** în figura 1.17 va fi validată. Drept consecinţă, impulsul de scriere aplicat pe intrarea WRITE se va propaga doar prin poarta SI menţionată, transformându-se în impuls de tact pentru toţi cei 32 de bistabili ce compun locaţia i (registrul R_i). Ca urmare, data prezentă pe intrările DI_{31-0} va fi scrisă în locaţia i (în registrul R_i).

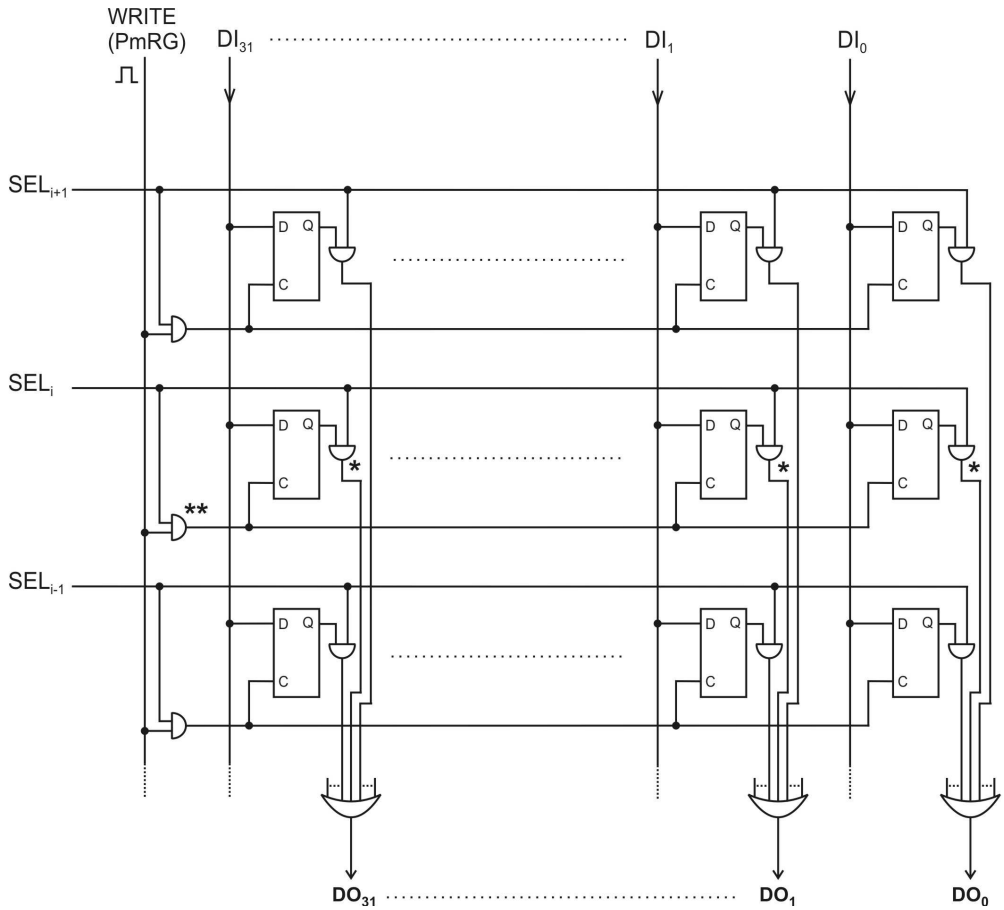


Fig 1.17 Structura planului de memorie M

1.5.2 Registre generale implementate în structuri multiport

Din punctul de vedere al performanţelor, implementarea registrelor generale într-o structură uniport prezintă anumite dezavantaje. Structura uniport utilizează o singură sursă de adresare, atât pentru operaţiile de citire cât si pentru cele de scriere (dispune de un singur set de intrări de adresă). Prin urmare, într-o anumită perioadă de tact procesor, un singur registru poate fi, fie scris, fie citit.

Procesoarele moderne prezintă un grad înalt de paralelism în execuţia operaţiilor interne. Pentru a susţine acest paralelism, setul de registre generale trebuie să permită accese multiple la registrele din componenţa sa, în aceeaşi perioadă de tact (accese concurente). De exemplu, în faza de execuţie a instrucţiunii ADD R3, R1, R2 (format cu trei adrese, vezi 1.3!), se execută operaţia $R_3 \leftarrow R_1+R_2$. Pentru a efectua această operaţie într-o singură perioadă de tact procesor, setul de registre trebuie să permită trei accese simultane:

-citire R_1 (operand 1)
-citire R_2 (operand 2)
-scriere R_3 (rezultat)

Aceste trei accese simultane nu sunt posibile în structura uniport dar devin posibile în cazul implementării registrelor generale într-o structură multiport (în cazul exemplificat vor fi necesare 2 porturi de citire şi unul de scriere). Implementarea celor 8 registre generale (pe 32 biţi) într-o structură multiport, cu două porturi de citire şi unul de scriere, este prezentată în fiura 1.18. În această figură s-au utilizat următoarele notaţii:

AS_{2-0} -adresă scriere; selectează registrul general ce va fi scris.
DI_{31-0} -intrările de date; pe aceste linii se aplică informaţia ce urmează a fi scrisă în registrul selectat cu adresa AS_{2-0}.
$AC_2A \div AC_0A$ -adresă de citire aferentă portului A; selectează registrul ce va fi citit pe portul A.
$DO_{31}A \div DO_0A$ -ieşirile de date aferente portului A; pe aceste ieşiri se va obţine cuvântul de 32 biţi citit din registrul selectat cu adresa $AC_2A \div AC_0A$.

$AC_2B \div AC_0B$ -adresă de citire aferentă portului B; selectează registrul ce va fi citit pe portul B.
$DO_{31}B \div DO_0B$ -ieşirile de date aferente portului B; pe aceste ieşiri se va obţine cuvântul de 32 biţi citit din registrul selectat cu adresa $AC_2B \div AC_0B$.
PmRi -„primeşte R_i”; comanda de scriere în registrul R_i.

Pentru efectuarea operaţiei $R_3 \leftarrow R_1+R_2$, se vor realiza trei accese simultane la setul de registre generale din figura 1.18:

- R_1 citit pe portul A ($AC_2A \div AC_0A=001_2$)
- R_2 citit pe portul B ($AC_2B \div AC_0B=010_2$)
- R_3 scris pe portul de scriere ($AS_{2-0}=011_2$)

Ne putem imagina faptul că, în structura procesorului, datele $DO_{31}A \div DO_0A$ (operandul1) şi datele $DO_{31}B \div DO_0B$ (operandul 2) sunt aplicate pe cele două intrări ale unităţii ALU (*Arithmetic and Logic Unit*), că pentru unitatea ALU se activează comanda SUM (de adunare) şi că rezultatul obţinut pe ieşirea unităţii ALU este aplicat pe intrările de date $DI_{31} \div DI_0$ aferente portului de scriere din figura 1.10, activând suplimentar şi comanda WRITE. Dacă cele trei seturi de adrese din figura 1.18 sunt setate pe valorile precizate mai sus, este evident că efectuarea operaţiei $R_3 \leftarrow R_1+R_2$, într-o singură perioadă de tact procesor, devine posibilă (pe structura multiport).

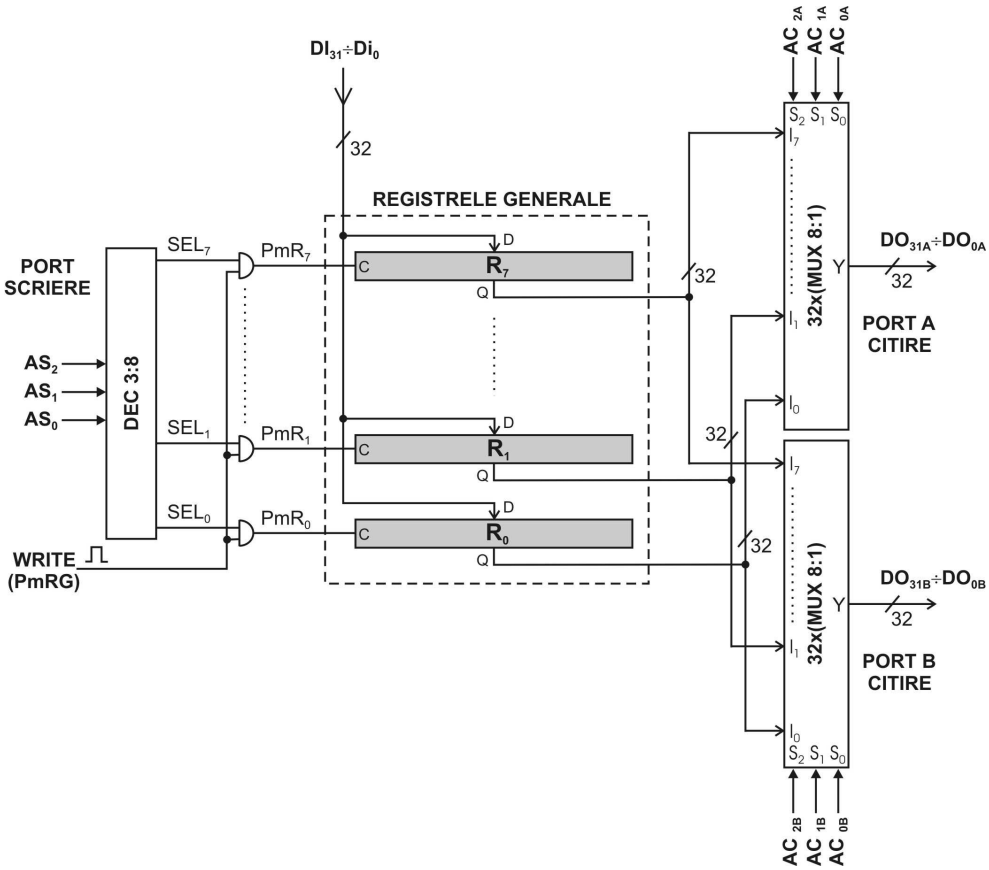


Fig 1.18 Structura multiport aferentă unui set de 8 registre generale pe 32 biţi

Observaţie:
Pentru a satisface gradul înalt de paralelism intern impus procesoarelor moderne, setul de registre generale se implementează de regulă cu mai multe porturi. De exemplu, microprocesorul RISC Motorola 88110 conţine două seturi a câte 32 registre generale:

$R_{31} \div R_0$ *-registre generale pentru numere întregi pe 32 biţi (virgulă fixă)*
$X_{31} \div X_0$ *-registre generale pentru numere reprezentate în virgulă flotantă (pe 80 biţi)*

Ambele seturi de registre au câte 8 porturi: 6 porturi de citire şi 2 porturi de scriere.

1.6 Memoria internă

Memoria internă se mai numeşte şi **memorie principală** sau **memorie centrală**. După cum am precizat la 1.4, memoria internă reprezintă nivelul 2 din structura ierarhizată a memoriei. Acest nivel se implementează cu circuite integrate specializate (circuite de memorie). Cele mai utilizate circuite sunt circuitele RAM-dinamic. Celula de memorare RAM-dinamic are o structură foarte simplă (o simplă joncţiune implementată în tehnologie MOS care funcţionează ca un condensator). Dacă condensatorul este încărcat (joncţiunea este încărcată) cu o sarcină electrică Q, semnificativ diferită de zero, spunem că celula memorează un bit de valoare ”1”; dacă condensatorul este descărcat (Q=0), spunem că celula memorează un bit de valoare ”0”.

Structura extrem de simplă a celulei RAM-dinamic va conduce la o densitate mare de celule pe unitatea de suprafaţă (în pastila de siliciu) şi prin urmare se obţin circuite de memorie de capacitate foarte mare. Acesta este marele avantaj al tehnologiei RAM-dinamic.

Principalul dezavantaj al tehnologiei RAM-dinamic este reprezentat de necesitatea regenerării periodice a informaţiilor. Condensatorul (joncţiunea MOS), din punct de vedere electric, nu poate fi izolat perfect faţă de linia de bit asociată (prin intermediul căreia se scrie sau se citeşte celula). Prin urmare, „apar curenţii de fugă” care conduc la un efect nedorit: o celulă încărcată cu o sarcină Q semnificativ diferită de zero (care memorează ”1”) va fi afectată, în timp, de un proces de descărcare a sarcinii Q iar dacă sarcina Q scade la un nivel apropiat de zero, celula va trece din ”1” în ”0”. Pentru a evita această trecere din ”1” în ”0”, este necesară regenerarea periodică a informaţiilor. Practic, toate celulele care memorează ”1” trebuie reîncărcate cu sarcină Q la intervale regulate de timp. Prin reîncărcare cu sarcină Q, trebuie să înţelegem reîncărcarea condensatorului (joncţiunii MOS) cu Q la valoarea maximă.

Memoria internă a calculatorului conţine şi un plan de memori ROM (*Read-Only Memory*) de capacitate mică. La calculatorul IBM-PC în acest plan de ROM este „ars” BIOS-ul (*Basic Input-Output System*).

De ce ar fi necesar un plan de ROM ? Pentru că, la punerea sub tensiune a calculatorului, în RAM nu există informaţie utilă. Memoria RAM este volatilă iar la punerea sub tensiune, în RAM acţionează hazardul (anumite celule se setează pe ”1” iar altele pe ”0”).

Procesorul „nu ştie” altceva decât să proceseze instrucţiuni (să execute programe). Imediat după punerea sub tensiune, în RAM nu există nici-un program. Prin urmere,

calculatorul nu-si poate începe activitatea decât prin execuţia unui program din ROM (dintr-un plan de memorie nevolatilă). Practic, la punerea sub tensiune a calculatorului, din sursa de alimentare, se generează un semnal de iniţializare (RESET), similar celui activat din butonul RESET de pe panoul frontal. Semnalul RESET va iniţializa procesorul şi celelalte blocuri *hardware*. Pe lângă multe altele, iniţializarea procesorului înseamnă şi încărcarea în registrul PC (*Program Counter*) a unei constante (adresa unei locaţii din planul de ROM). La această adresă se va afla codul primei instrucţiuni pe care procesorul, după punerea calculatorului sub tensiune, o va extrage şi o va executa. Iniţializarea şi restartarea sistemului de calcul decurg analog atunci când resetarea acestuia este realizată de către operator, prin acţionarea butonului RESET.

Procesor didactic. Structură şi proiectare

În figura 2.1 se prezintă schema bloc a calculatorului, schemă introdusă şi descrisă succint în capitolul introductiv (paragraful 2.1). Această structură redă într-o formă mult mai compactă (şi mai uzitată astăzi) „structura calculatorului cu program memorat” definită de către von Neumann în anul 1945, pe baza celor cinci criterii enunţate în „Prima schiţă de raport asupra calculatorului EDVAC”.

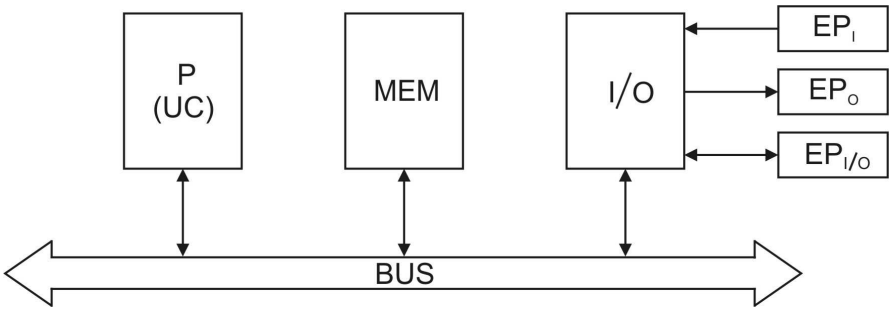


Fig 2.1 Schema bloc a calculatorului

Elementele prezente în figura 2.1 sunt:

- P –procesor, denumit uneori şi unitate centrală (UC).
- MEM –memoria principală, denumită uneori şi memorie internă sau memorie centrală. Memoria principală este implementată cu circuite integrate specializate (circuite de memorare).
- I/O –blocul interfeţelor de intrare-ieşire (*input-output*). Acest bloc conţine interfeţele de intrare-ieşire aferente perifericelor. Fiecare periferic are o interfaţă dedicată. Aceste interfeţe dedicate se mai numesc şi *controller*-e de intrare-ieşire (*controller* de disc, *controller* de tastatură, *controller* de imprimantă etc.).

- BUS –busul sistemului. Conţine trei tipuri de linii de bus: de date, de adrese şi de comenzi. Pe acest bus se efectuează transferurile de informaţii dintre cele trei blocuri din structura calculatorului. Transferurile care au loc frecvent pe acest bus în timpul funcţionării calculatorului sunt:

  - *Fetch* instrucţiune (citirea codului instrucţiunii din memorie şi încărcarea acestuia în procesor). După *fetch* urmează decodificarea şi execuţia instrucţiunii.

  - *Fetch* operand (citirea operandului unei instrucţiuni din memorie şi incărcarea acestuia în procesor cu scopul de a fi utilizat în faza de execuţie a instrucţiunii).

  - Memorare rezultat (depunerea rezultatului produs de instrucţiune în memorie). Memorarea rezultatului este ultima operaţie din cadrul fazei de execuţie a instrucţiunii.

  - Citirea unui port de intrare printr-o instrucţiune IN (*INput*).

  - Scrierea unui port de ieşire printr-o instrucţiune OUT (*OUTput*). Porturile de intrare/ieşire sunt registre speciale (de date şi respectiv de control) plasate în *controller*-ele de intrare-ieşire.

  Primele trei transferuri exemplificate sunt transferuri procesor-memorie. În primele două memoria reprezintă sursa iar procesorul reprezintă destinaţia datelor transferate; în al treilea transfer procesorul devine sursă şi memoria destinaţie. În ultimele două transferuri sunt implicate procesorul şi respectiv blocul I/O.

- $EP_I/EP_O/EP_{I/O}$ –echipamente periferice de intrare/ieşire/intrare-ieşire.

Sarcina globală a calculatorului este aceea de a executa programe de aplicaţii şi de a furniza rezultatele diverselor probleme transpuse de către programatori în astfel de programe. Fiecare dintre unităţile evidenţiate în figura 2.1 este capabilă să execute anumite operaţii specifice (din cadrul sarcinii globale):

- procesorul citeşte din memorie şi execută, una câte una, instrucţiunile unui program aflat în execuţie.

- memoria stochează şi ţine la dispoziţia procesorului instrucţiuni (programe) şi date (datele de intrare aferente diverselor programe şi respectiv rezultatele produse de aceste programe).

- blocul I/O asigură legătura (comunicaţia) între calculator şi echipamentele periferice conectate la acesta.

2.1 Schema bloc a procesorului didactic

Pentru procesorul didactic pe care-l vom proiecta alegem o structură simplă, bazată pe acumulator (fig. 2.2). Acumulatorul este un registru special din procesor în care se depun rezultatele (toate instrucţiunile aritmetico-logice depun rezultatul în acumulator). De

asemenea, în cazul instrucţiunilor cu doi operanzi, primul operand va fi preluat întotdeauna din acumulator.

În structura procesorului didactic pot fi identificate trei busuri interne:

SBUS –busul sursă (pentru intrarea S în ALU); preia operandul 1 din acumulator (registrul A) şi îl aplică pe intrarea S a unităţii ALU.

DBUS –busul destinaţie (pentru intrarea D în ALU); preia operandul 2 şi îl aplică pe intrarea D a unităţii ALU.

RBUS –busul rezultatelor; preia rezultatele de pe ieşirea R a unităţii ALU.

Observaţie:
Să notăm că denumirile de „bus sursă” şi respectiv „bus destinaţie” sunt fireşti la procesoarele care utilizează formatul de instrucţiune cu două adrese (figura 1.7). În cadrul unui astfel de format prima adresă reprezintă adresa operandului sursă (operandul 1) iar a doua adresă reprezintă adresa operandului destinaţie (operandul 2). Rezultatul se va depune evident peste operandul destinaţie; prin depunere operandul destinaţie se pierde. Aceasta este de fapt motivaţia sintagmei „operand destinaţie”.

Utilizarea denumirilor de „bus sursă” şi respectiv „bus destinaţie” nu mai sunt tot atât de fireşti la procesorul didactic datorită faptului că aici vom utiliza formatul cu o singură adresă. Am preferat totuşi să utilizăm această terminologie (în principiu consacrată) şi aici.

Cele 3 busuri interne interconectează următoarele elemente din structura procesorului didactic:

ALU –unitatea aritmetico-logică (*Arithmetic and Logic Unit*). Este un bloc pur combinaţional care execută 4 operaţii aritmetico-logice, pe baza următoarelor comenzi:

- *SUM* –comanda de adunare a celor 2 operanzi aplicaţi pe cele două intrări ale unităţii ALU. Operandul 1 este preluat de pe SBUS, operandul 2 este preluat de pe DBUS iar rezultatul va fi emis pe RBUS; emiterea rezultatului pe RBUS se produce prin activarea comenzii *PdALU* („predă ALU”).
- *AND* –SI logic între cei 2 operanzi preluaţi de pe SBUS şi respectiv DBUS.
- *OR* –SAU logic între cei 2 operanzi preluaţi de pe SBUS şi respectiv DBUS.
- *XOR* –SAU EXCLUSIV între cei 2 operanzi preluaţi de pe SBUS şi respectiv DBUS.

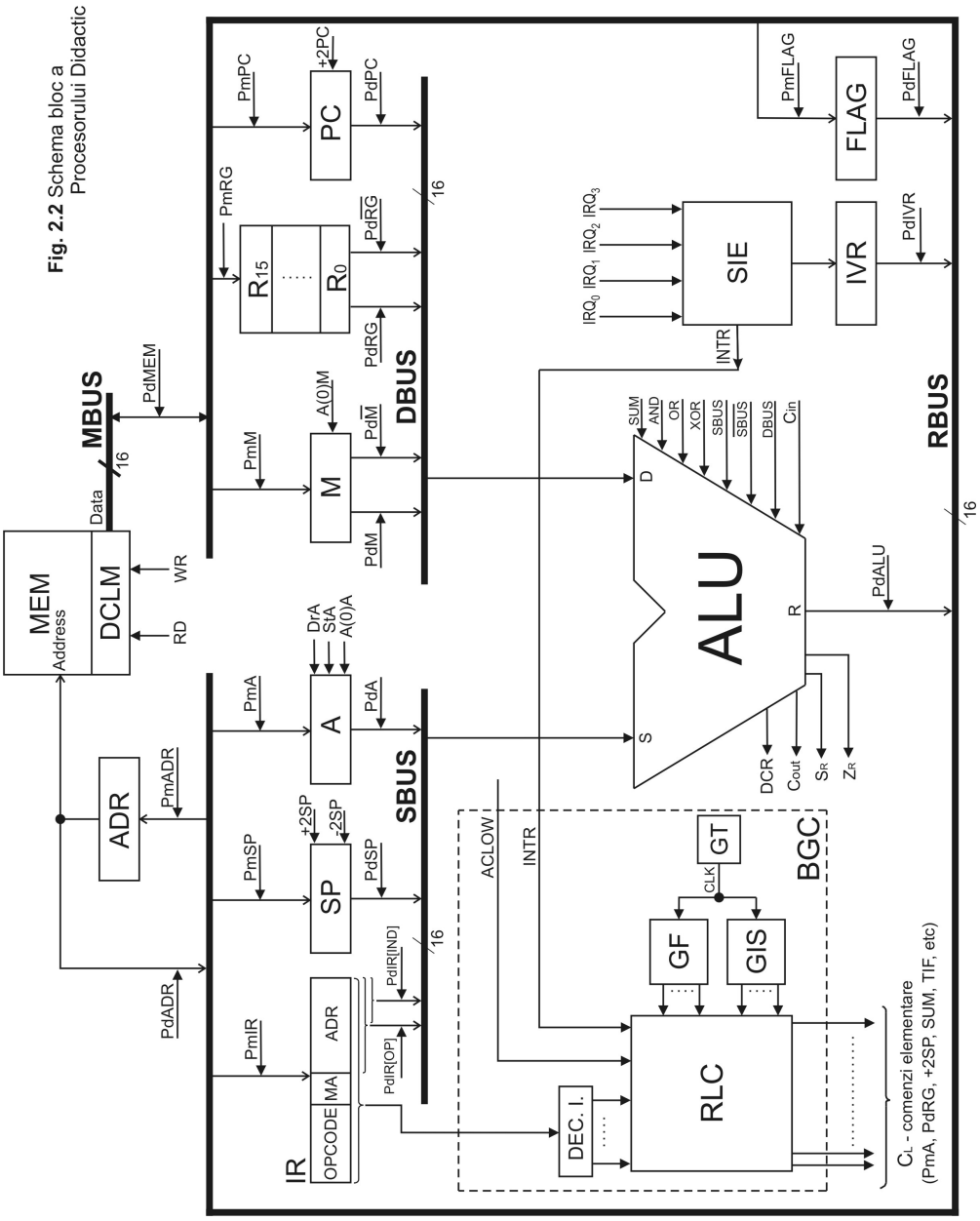


Fig 2.2 Schema bloc a Procesorului Didactic

Pe lângă cele patru comenzi (operaţii) aritmetico-logice, ALU execută şi următoarele trei comenzi (operaţii) simple:

- *SBUS* –transferă (nemodificat) operandul preluat de pe SBUS pe ieşirea ALU.

- $\overline{SBUS}$ –inversează, bit cu bit, operandul preluat de pe SBUS şi îl transferă pe ieşirea ALU (complementul faţă de 1 al operandului preluat de pe SBUS va fi emis pe ieşirea ALU).

- *DBUS* –transferă (nemodificat) operandul preluat de pe DBUS pe ieşirea ALU.

Tot la unitatea ALU se disting următoarele semnale de condiţii:

- Z_R –semnalizează rezultat egal cu zero pe ieşirea ALU (dacă Z_R=1) şi respectiv diferit de zero (dacă Z_R=0).

- S_R –semnul rezultatului de pe ieşirea ALU; rezultat pozitiv, dacă S_R=0 şi respectiv negativ, dacă S_R=1).

- DCR –depăşirea capacităţii de reprezentare. Cu alte cuvinte, rezultatul depăşeşte domeniul de reprezentare (este incorect), dacă DCR=1 şi respectiv se află în domeniul de reprezentare (este corect), dacă DCR=0.

- C_{out} –transport de ieşre (*Carry Out*). Semnalul (condiţia) C_{out} se activează (C_{out}=1) la adunare dacă se obţine un transport de ieşire din rangul cel mai semnificativ (rangul de semn). Să notăm că scăderea se execută tot printr-o adunare (se adună la descăzut complementul faţă de 2 al scăzătorului). Din punct de vedere matematic însă, C_{out}=1 la scădere are semnificaţia de împrumut (*borrow*) pe care rangul de semn îl revendică rangului superior (rang care de altfel nu există).

  Cele patru semnale de condiţii (Z_R/S_R/DCR/C_{out}) sunt semnale de ieşire din ALU şi vor fi utilizate la setarea (poziţionarea) *flag*-urilor de condiţii (bistabilii Z/S/V/C) din registrul FLAG. Să notăm că *flag*-urile de condiţii trebuie setate (în acord cu rezultatul obţinut în ALU) la toate instrucţiunile aritmetice şi respectiv logice.

- C_{in} –transport de intrare (*Carry In*). Este în realitate un semnal de intrare în ALU (intrarea de transport în rangul zero al sumatorului din componenţa ALU). Este deci o intrare de comandă pentru ALU (dacă se activează simultan comenzile *SUM* şi C_{in} se pot realiza incrementări cu 1 în ALU).

IR –registrul instrucţiunii (*Instruction Register*). Procesarea oricărei instrucţiuni din cadrul unui program începe cu citirea codului acesteia din memorie şi încărcarea acestui cod în IR (*fetch*-ul instrucţiunii).

SP –registrul pointer de stivă (*Stack Pointer*). Stiva este o zonă de memorie rezervată prin program pentru memorarea temporară a unor date. Registrul SP este utilizat pentru adresarea locaţiilor stivei.

A –acumulator (*Accumulator*). În cazul instrucţiunilor cu doi operanzi, registrul acumulator conţine întotdeauna primul operand; tot în acumulator se depune şi rezultatul. Rezultă că în codul instrucţiunii va trebui codificată doar adresa celui de-al doilea operand. Prin urmare, introducerea registrului A în structura procesorului permite utilizarea formatului de instrucţiune cu o singură adresă (simplificare).

M –registru de manevră (registru tampon de date). Este inaccesibil programatorului (nu există instrucţiuni care să facă referinţă la M). Registrul M este utilizat la nivel *hardware* pentru memorarea temporară a unor date. De asemenea, prin registrul M, procesorul se interfaţează cu memoria (în M se încarcă de regulă operanzii citiţi din memorie).

RG –setul de registre generale. Conţine 16 registre ($R_0 \div R_{15}$) pe 16 biţi. Registrele generale sunt alocate, în faza de compilare, operanzilor frecvent utilizaţi (variabilelor frecvent utilizate) în cadrul programului. Accesul la un registru este mult mai rapid decât accesul la o locaţie de memorie.

ADR –registrul de adrese (*ADdress Register*). Înaintea unei operaţii de scriere în sau citire din memorie, în ADR trebuie încărcată adresa locaţiei ce urmează a fi accesată (scrisă sau citită).

FLAG –registrul de *flag*-uri. Conţine *flag*-urile (bistabilii) de condiţii:

- C –*carry* (transport de ieşire din rangul cel mai semnificativ)
- Z –*zero* (rezultat egal cu zero pe ieşirea ALU)
- S –*sign* (semnul rezultatului de pe ieşirea ALU)
- V –*overflow* (rezultat în afara domeniului de reprezentare, adică incorect)

IVR –registrul vectorului de întrerupere (*Interrupt Vector Register*). La activarea unei întreruperi externe (de către un periferic) procesorul intră

în faza de întrerupere. Faza de întrerupere are rolul de a iniţia tratarea întreruperii activate. Prima operaţie realizată în cadrul fazei de întrerupere este achitarea întreruperii: la cererea de întrerupere INTR (*INTerrupt Request*) procesorul răspunde cu semnalul de achitare INTA (*INTerrupt Acknowledge*). Sesizând acivarea semnalului de achitare INTA, *controller*-ul I/O constată că tratarea întreruperii a fost deja iniţiată de către procesor şi drept consecinţă va dezactiva cererea INTR.

Tot în faza de întrerupere, sistemul de întreruperi şi excepţii (SIE) va genera vectorul de întrerupere asociat perifericului care a activat întreruperea şi-l va încărca în registrul IVR. Acest vector va fi utilizat ca pointer în tabela IVT (*Interrupt Vector Table*). Din tabelă se va obţine adresa de start a *handler*-ului de întrerupere (program din componenţa sistemului de operare care va face, perifericului, serviciul solicitat). Fiecare periferic are propriul *handler* de întrerupere.

SIE –sistemul de întreruperi şi excepţii. La tratarea întreruperilor şi excepţiilor contribuie atât *hardware*-ul cât şi *software*-ul (sistemul de operare). *Hardware*-ul implicat este reprezentat de blocul SIE şi de registrul IVR. SIE este acţionat de patru linii externe de întrerupere $IRQ_0 \div IRQ_3$ (***I**nterrupt **R**e**Q**uest*) şi de anumite excepţii cum ar fi:

CIL -cod ilegal (instrucţiune ilegală)
ACLOW -căderea tensiunii de alimentare

Excepţiile se mai numesc şi întreruperi interne deoarece sunt cauzate de anomalii (erori) detectate în procesor.
SIE generează cererea globală de întrerupere INTR.

BGC –blocul de generare a comenzilor. Este unitatea de control a procesorului (*Control Unit*). BGC generează toate semnalele de comandă prin care se iniţiază şi se execută operaţii elementare. O operaţie elementară este operaţia care poate fi executată într-o singură perioadă de tact. De exemplu, pentru execuţia unei instrucţiuni de adunare sunt necesare următoarele operaţii elementare:

- citirea din memorie a operanzilor şi încărcarea acestora în procesor.
- aplicarea operanzilor pe intrările ALU, efectuarea operaţiei de adunare şi depunerea rezultatului la destinaţie.
- setarea *flag*-urilor de condiţii (bistabilii C/Z/S/V din registrul FLAG) în acord cu rezultatul adunării.

Toate aceste operaţii sunt efectuate prin activarea unor comenzi elementare. Comenzile trebuie activate într-o succesiune bine definită. Referinţa de timp o dă semnalul de tact al procesorului. Prin urmare, fiecare comandă elementară se va activa pe o anumită perioadă de tact.

Succesiunea acestor comenzi trebuie clar definită astfel încât să conducă la succesiunea corectă de operaţii elementare. În concluzie, execuţia unei instrucţiuni se întinde pe una sau mai multe perioade de tact şi constă dintr-o succesiune bine definită de operaţii elementare.

BGC poate fi implementat în două variante:

- cablat (uzual la procesoarele RISC –*Reduced Instruction Set Computer*)
- microprogramat (uzual la procesoarele CISC –*Complex Instruction Set Computer*)

În variantă cablată, BGC conţine trei componente importante:

- RLC –reţeaua logică combinaţională
- GF –generatorul de faze. Ciclul instrucţiunii este format din mai multe faze (*fetch* instrucţiune, *fetch* operand, execuţie etc.) care se parcurg succesiv, în vederea procesării acesteia. Fiecare fază are asociat un semnal de fază iar semnalele de fază sunt generate de GF.
- GIS –generatorul impulsurilor de stare. O fază conţine o succesiune de impulsuri de stare. Durata unui impuls de stare este egală cu perioada de tact. Prin urmare, cu ajutorul semnalelor de fază şi a impulsurilor de stare se poate identifica fiecare perioadă de tact din ciclul instrucţiunii. În concluzie, GF şi GIS (blocuri secvenţiale) identifică fiecare perioadă de tact din ciclul instrucţiunii, iar RLC (bloc combinaţional) precizează setul de comenzi elementare care trebuie activat pe fiecare perioadă de tact.

Să observăm că BGC (mai precis, RLC din cadrul BGC) identifică instrucţiunea curentă (instrucţiunea de executat) cu ajutorul DEC.I. (decodificatorul instrucţiunii). Să mai observăm că BGC funcţionează pe baza semnalului de tact al procesorului generat de către generatorul de tact GT.

MEM –Memoria principală (denumită şi memorie internă sau centrală). Aici se încarcă programele în vederea lansării lor în execuţie.

DCLM –Dispozitivul de Comandă Locală aferent Memoriei. Este un automat specializat care controlează operaţiile de citire din şi respectiv de scriere în memorie solicitate de către procesor prin comenzile RD (*ReaD*) şi respectiv WR (*WRite*).

MBUS –Busul Memoriei (*Memory BUS*). Este un bus de date prin intermediul căruia se realizează transferurile de date procesor – memorie. MBUS este interconectat cu RBUS printr-o conexiune bidirecţională. La operaţiile de citire din memorie sensul transferului pe această conexiune trebuie să fie MBUS→RBUS iar la operaţiile de scriere în memorie sensul transferului trebuie să fie RBUS →MBUS. Sensul va fi stabilit prin comanda *PdMEM* („predă memorie"). *PdMEM*=1 validează sensul MBUS→RBUS (necesar la o operaţie de citire din memorie) iar *PdMEM*=0 validează sensul RBUS →MBUS (necesar la o scriere în memorie).

În schema bloc a procesorului didactic sunt evidenţiate o serie de comenzi logice elementare (C_L) cu ajutorul cărora se declanşează şi controlează operaţiile elementare. Procesarea unei instrucţiuni se face în „paşi mărunţi" (operaţii elementare succesive). Pe fiecare perioadă de tact se efectuează una sau mai multe operaţii elementate. În schemă pot fi identificate următoarele tipuri de comenzi:

- *PmR* –„primeşte registru". Această comandă încarcă în registrul specificat informaţiile prezente pe intrările de date ale acestuia. Să exemplificăm câteva comenzi din această categorie:
    - *PmA* –„primeşte A" (încărcare paralelă în acumulator)
    - *PmPC* –„primeşte PC" (încărcare paralelă în *program counter*)
    - *PmIR* –„primeşte IR" (încărcare paralelă în registrul instrucţiunii)
- *PdR* –„predă registru". Această comandă emite informaţiile memorate în registrul specificat pe busul la care acesta este conectat. Să exemplificăm câteva comenzi din această categorie:
    - *PdIVR* –„predă IVR" (emite pe RBUS vectorul de întrerupere din IVR)
    - *PdIR[IND]* –„predă indexul din IR" (emite INDEX-ul codificat în codul instrucţiunii curente pe SBUS).
    - *PdALU* –„predă ALU" (emite pe RBUS rezultatul prezent pe ieşirea ALU)
- Comenzi de incrementare/decrementare. Se aplică registrelor care trebuie să efectueze astfel de operaţii. Prezentăm câteva exemple:
    - *+2PC* –„plus 2 PC" (incrementare cu 2 a registrului PC)
    - *+2SP* –„plus 2 SP" (incrementare cu 2 a registrului SP)
    - *–2SP* –„minus 2 SP" (decrementare cu 2 a registrului SP)

- Comenzi de iniţializare. Aceste comenzi se aplică unor registre şi au ca efect încărcarea constantei zero în registrul specificat. Prezentăm câteva exemple:

  - *A(0)A* –„adu la zero A” (resetare registru acumulator)
  - *A(0)M* –„adu la zero M” (resetare registru M)

- Comenzi pentru ALU. Prin aceste comenzi se stabilesc operaţiile efectuate în ALU. Comenzile din această categorie sunt: *SUM*, *AND*, *OR*, *XOR*, *SBUS*, $\overline{SBUS}$, *DBUS* şi C_{in}. Toate aceste comenzi au fost descrise mai sus (blocul ALU).

- Comenzi pentru memorie:

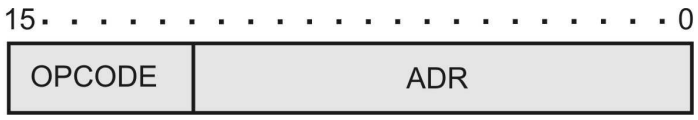
  - *RD* –citire din memorie
  - *WR* –scriere în memorie

  Să menţionăm că adresa locaţiei ce urmează a fi accesată (pentru citire sau scriere) trebuie să fie prezentă (încărcată anterior) în ADR. Comanda *PdMEM* a fost prezentată mai sus (vezi descrierea MBUS-ului).

2.2 Proiectarea setului de instrucţiuni

În acest paragraf vom defini un set de instrucţiuni pentru procesorul didactic a cărui structură a fost prezentată la 2.1 (figura 2.2). Va fi un set de instrucţiuni redus dar reprezentativ, în sensul că va conţine toate instrucţiunile de bază (cele mai uzuale şi mai importante) prezente în seturile de instrucţiuni aferente procesoarelor de uz general. Setul redus de instrucţiuni va conduce desigur la o reducere a complexităţii unităţii de control şi implicit la o înţelegere mai uşoară a metodologiei de proiectare a acesteia.

Procesorul didactic introdus la 2.1 este un procesor pe 16 biţi, orientat pe acumulator (structură tipică pentru procesoarele de complexitate redusă). În consecinţă, vom încerca să codificăm instrucţiunile pe 16 biţi şi să utilizăm (tot în ideia simplificării proiectului) formatul de lungime fixă. Structura bazată pe acumulator ne permite utilizarea formatului cu o singură adresă (fig 2.3) deoarece, în cazul instrucţiunilor cu doi operanzi, primul operand va fi preluat din acumulator (referit implicit) iar rezultatul va fi depus tot în acumulator; doar operandul doi va fi referit explicit printr-o adresă codificată în codul instrucţiunii.



OPCODE –codul operaţiei (codul propriu-zis al instrucţiunii)
ADR –**ADR**esa operandului sau adresa de salt (la instrucţiunile de salt)

Fig 2.3 Structura de principiu a formatului cu o singură adresă

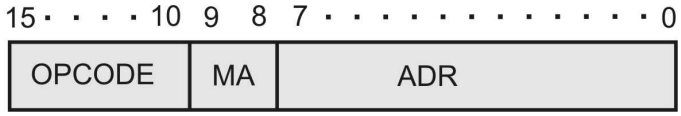
Setul de instrucţiuni aferent procesorului didactic va fi format din două clase de instrucţiuni:

- IOP –**I**nstrucţiuni cu **Op**erand. Sunt instrucţiuni care referă explicit operandul (operandul este adresat explicit cu câmpul ADR din codul instrucţiunii)

- ID –**I**nstrucţiuni **D**iverse. Sunt instrucţiuni fără operanzi sau instrucţiuni care referă implicit operandul. Referinţa la operand nu va fi explicită (prin câmpul ADR din codul instrucţiunii); acesta va fi referit implicit, prin câmpul OPCODE.

2.2.1 Formatul instrucţiunilor din clasa IOP

Formatul instrucţiunilor IOP este prezentat în figura 2.4.



OPCODE –codul operaţiei (codul propriu-zis al instrucţiunii)
MA –**M**od de **A**dresare (utilizat pentru localizarea operandului)
ADR –**ADR**esă (utilizată împreună cu MA pentru localizarea operandului)

Fig 2.4 Formatul instrucţiunilor din clasa IOP

Câmpurile MA şi ADR permit localizarea operandului referit explicit de instrucţiune. În cazul instrucţiunilor de salt cele două câmpuri permit localizarea instrucţiunii pe care se va face saltul. În consecinţă, adresa operandului din cazul general, devine adresă de salt în cazul instrucţiunilor de salt. Cei doi biţi din componenţa câmpului MA (biţii 9 şi respectiv 8 din codul instrucţiunii) ne permit să codificăm patru moduri de adresare (tabelul 2.1).

Câmpul MA		Mod de adresare
Bit 9	Bit 8	
0	0	Imediat (AM)
0	1	Direct (AD)
1	0	Indirect (AI)
1	1	Indexat (AX)

Tabelul 2.1 Codificarea modurilor de adresare

Localizarea operandului în cadrul celor patru moduri de adresare se face pornind de la câmpurile MA şi ADR, în conformitate cu schemele de principiu prezentate în figura 2.5.

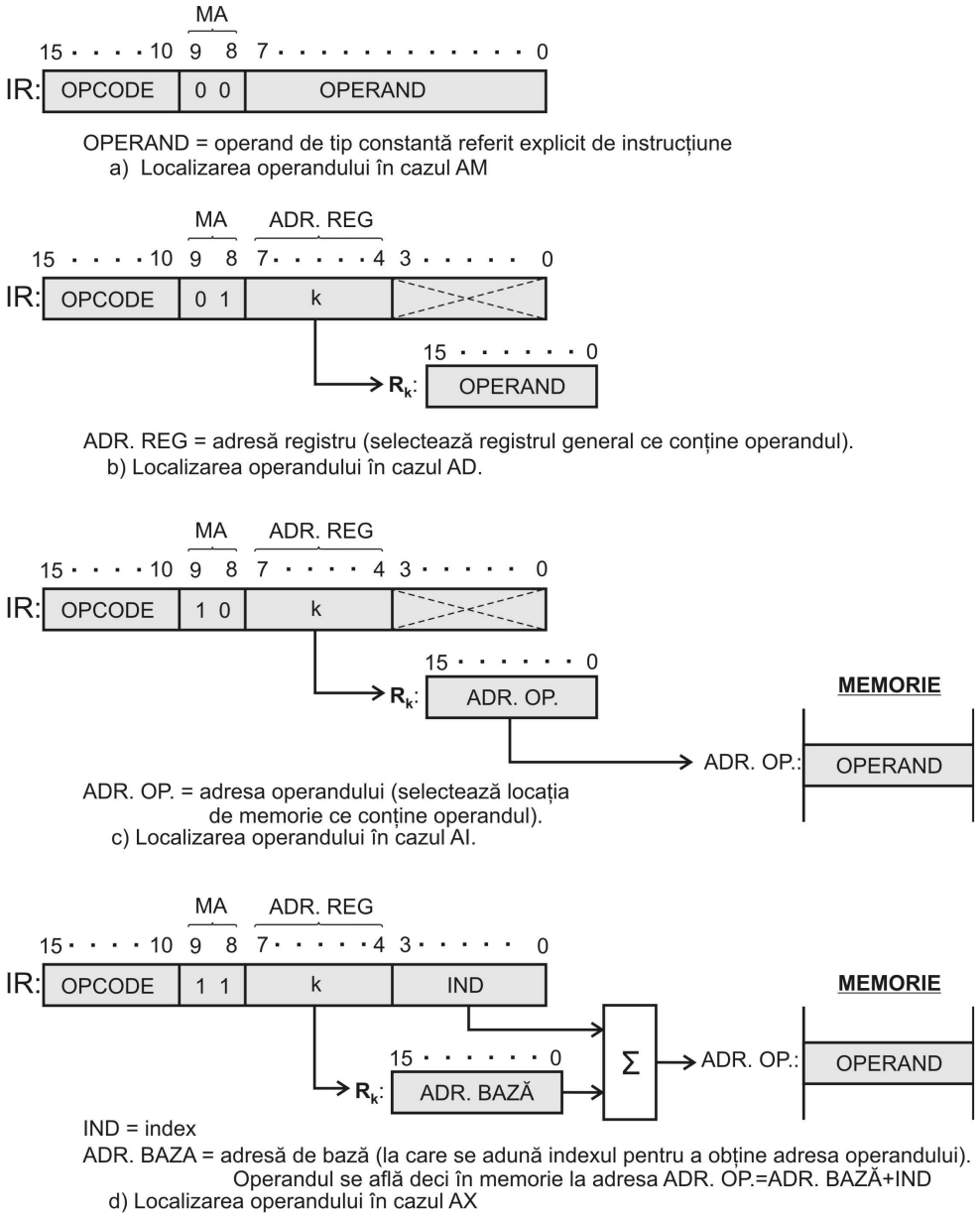


Fig 2.5 Localizarea operandului în cadrul celor patru moduri de adresare

Observaţie:
Să notăm că atunci când ne referim la codul instrucţiunii curente ne referim de fapt la conţinutul registrului IR. În momentul extragerii instrucţiunii din memorie (instruction fetch) codul acesteia va fi încărcat în IR şi rămâne în IR până la finalizarea execuţiei acesteia. Cu alte cuvinte codul unei instrucţiunii este prezent în IR pe toată durata procesării instrucţiunii respective. El este „evacuat" din IR doar prin încărcarea noului cod (fetch-ul următoarei instrucţiuni). Din acest motiv, când ne referim la anumiţi biţi din codul instrucţiunii curente, facem de fapt referire la biţi din registrul IR. Câmpul MA este de pildă format din biţii IR_9 *şi* IR_8.

Pentru câmpul opcode am rezervat 6 biţi ($IR_{15\div10}$). Pe de altă parte, am definit două clase de instrucţiuni, IOP şi respectiv ID. Bitul cel mai semnificativ (IR_{15}) va selecta clasa:

- IR_{15}=0 defineşte clasa IOP
- IR_{15}=1 defineşte clasa ID

Rămân 5 biţi de OPCODE propriu-zis ($IR_{14 \div 10}$) pentru clasa IOP. În consecinţă vom putea codifica maxim 2^5=32 instrucţiuni în clasa IOP. Instrucţiunile din clasa IOP sunt prezentate în tabelul 2.2, în ordinea în care acestea au fost codificate. Notaţiile utilizate în tabelul 2.2 au următoarele semnificaţii:

src –operand sursă localizat în conformitate cu modul de adresare codificat în instrucţiune

dest –operand destinaţie localizat în conformitate cu modul de adresare codificat în instrucţiune

reg –registru general specificat în instrucţiune (la instrucţiuni care operează doar cu mod de adresare <u>direct</u>)

adr –adresă de salt calculată în conformitate cu modul de adresare codificat în instrucţiune

Primele trei combinaţii binare ale câmpului OPCODE definesc instrucţiuni *load* care diferă între ele doar prin faptul că specifică regiştrii destinaţie diferiţi. Instrucţiunile ADD (*ADDition*) şi SUB (*SUBstraction*) sunt instrucţiuni de adunare şi respectiv scădere (într-un cuvânt aritmetice) şi ca atare, pe lângă operaţia aritmetică efectuată, vor seta şi *flag*-urile de condiţii din registrul FLAG în acord cu rezultatul obţinut. Instrucţiunea CMP (*CoMPare*) este practic o instrucţiune SUB care nu depune rezultatul la destinaţie (compararea se face prin scădere, fără depunerea rezultatului dar cu setarea *flag*-urilor de condiţii în acord cu rezultatul). Instrucţiunile AND, XOR (*eXclusive OR*) şi OR sunt instrucţiuni logice în care se execută operaţia logică specificată, se depune rezultatul la destinaţie (în acumulator) şi se setează *flag*-urile de condiţii în acord cu rezultatul. Să observăm că la instrucţiunile logice, indiferent de valorile operanzilor, nu poate rezulta C_{out}=1 şi respectiv DCR=1. În consecinţă niciodată nu se va putea obţine C=1 sau V=1 în urma unei instrucţiuni logice. Prin urmare, la aceste instrucţiuni se impune setarea (în acord cu rezultatul) doar a *flag*-urilor Z şi S.

OPCODE						INSTRUCŢIUNE	
IR_{15}	IR_{14}	IR_{13}	IR_{12}	IR_{11}	IR_{10}		
0	0	0	0	0	0	LD A,*src*	;A←*src* (*load* A)
0	0	0	0	0	1	LD SP,*src*	;SP←*src* (*load* SP)
0	0	0	0	1	0	LD FLAG,*src*	;FLAG←*src* (*load* FLAG)
0	0	0	0	1	1	ADD A,*src*	;A←A+*src* şi setare *flag*-uri de condiţii
0	0	0	1	0	0	SUB A,*src*	;A←A-*src* şi setare *flag*-uri de condiţii
0	0	0	1	0	1	CMP A,*src*	;A-*src* doar pentru setare *flag*-uri de condiţii
0	0	0	1	1	0	AND A,*src*	;A←A ∧ *src* şi setare *flag*-uri de condiţii
0	0	0	1	1	1	XOR A,*src*	;A←A ⊕ *src* şi setare *flag*-uri de condiţii
0	0	1	0	0	0	OR A,*src*	;A←A ∨ *src* şi setare *flag*-uri de condiţii
0	0	1	0	0	1	INC *dest*	;*dest*←*dest*+1 şi setare *flag*-uri de condiţii
0	0	1	0	1	0	DEC *dest*	; *dest*←*dest*-1 şi setare *flag*-uri de condiţii
0	0	1	0	1	1	PUSH *reg*	;STIVĂ←*reg* (salvare registru în stivă)
0	0	1	1	0	0	POP *reg*	;*reg*←STIVĂ (restaurare registru din stivă)
0	0	1	1	0	1	ST *dest*,A	;*dest*←A (*store* A)
0	0	1	1	1	0	ST *dest*,SP	;*dest*←SP (*store* SP)
0	0	1	1	1	1	ST *dest*,FLAG	;*dest*←FLAG (*store* FLAG)
0	1	0	0	0	0	JC *adr*	;*if* C=1 *then* PC←*adr* (*jump if carry*)
0	1	0	0	0	1	JNC *adr*	;*if* C=0 *then* PC←*adr* (*jump if not carry*)
0	1	0	0	1	0	JZ *adr*	;*if* Z=1 *then* PC←*adr* (*jump if zero*)
0	1	0	0	1	1	JNZ *adr*	;*if* Z=0 *then* PC←*adr* (*jump if not zero*)
0	1	0	1	0	0	JP *adr*	;*if* S=0 *then* PC←*adr* (*jump if plus*)
0	1	0	1	0	1	JM *adr*	;*if* S=1 *then* PC←*adr* (*jump if minus*)
0	1	0	1	1	0	JV *adr*	;*if* V=1 *then* PC←*adr* (*jump if overflow*)
0	1	0	1	1	1	JNV *adr*	;*if* V=0 *then* PC←*adr* (*jump if not overflow*)
0	1	1	0	0	0	JMP *adr*	;PC←*adr* (salt necondiţionat)
0	1	1	0	0	1	CALL *adr*	;STIVĂ←PC, PC←*adr* (apel procedură)
0	1	1	0	1	0	—	
0	1	1	0	1	1	—	;Combinaţii neutilizate. Rămân disponibile
0	1	1	1	0	0	—	;pentru dezvoltări viitoare (la nevoie pot fi
0	1	1	1	0	1	—	;definite noi instrucţiuni în clasa IOP)
0	1	1	1	1	0	—	
0	1	1	1	1	1	—	

Tabelul 2.2 Codificarea instrucţiunilor din clasa IOP

Instrucţiunile INC (*INCrement*) şi respectiv DEC (*DECrement*) sunt instrucţiuni de incrementare/decrementare cu 1. Incrementarea/decrementarea operandului înseamnă în fond modificarea valorii acestuia. Modificarea valorii presupune o operaţie de scriere şi din acest motiv operandul nu va mai avea semnificaţia de operand sursă ci de operand destinaţie. Suplimentar, INC şi DEC sunt instrucţiuni aritmetice şi prin urmare vor seta şi *flag*-urile de condiţii în acord cu rezultatul incrementării/decrementării. Instrucţiunile PUSH şi respectiv POP salvează şi respectiv restaurează registrul general (specificat de către instrucţiune) în/din stivă. Aceste instrucţiuni operează (au sens) doar cu mod de adresare direct.

Următoarele trei instrucţiuni sunt instrucţiuni *store* care memorează registrul specificat (A/SP/FLAG) la destinaţia (*dest*) codificată explicit în instrucţiune. Să observăm că, raportat la instrucţiunile *load*, instrucţiunile *store* operează exact în sens invers. În esenţă, instrucţiunile *load* ne permit încărcarea operanzilor (datelor) din memorie în procesor iar instrucţiunile *store* ne permit evacuarea rezultatelor (care sunt tot date) din procesor în memorie.

Să mai observăm că instrucţiunile *load*/*store* cu mod de adresare registru direct devin instrucţiuni *move* deoarece operandul, în acest caz, este conţinutul registrului general specificat. De exemplu, instrucţiunea LD A,R_{12} transferă conţinutul registrului R_{12} în acumulator ($A \leftarrow R_{12}$), iar instrucţiunea ST R_9,A transferă conţinutul acumulatorului în registrul R_9 ($R_9 \leftarrow A$); ambele sunt de fapt instrucţiuni *move* registru–registru.

Instrucţiunile JC, JNC, JZ, JNZ, JP, JM, JV şi JNV sunt instrucţiuni de salt condiţiunat. Toate aceste instrucţiuni testează o cundiţie şi dacă condiţia este îndeplinită fac salt la adresa de salt specificată. Condiţiile testate sunt de fapt cele patru *flag*-uri de condiţii: C –*carry*, Z –*zero*, S –*sign* şi respectiv V –*overflow*. Pe fiecare *flag* acţionează două instrucţiuni; una testează *flag*-ul pe fals iar cealaltă pe adevărat. Dacă condiţia testată este neândeplinită atunci saltul este inefectiv (se trece secvenţial la următoarea instrucţiune din program). Instrucţiunea JMP este o instrucţiune de salt necondiţionat; realizează întotdeauna (adică necondiţiunat) saltul la adresa specificată. Instrucţiunea CALL realizează apelul unei proceduri (subrutine) prin două operaţii succesive:

- salvează registrul PC în stivă (adresa de revenire în programul apelant, revenire care trebuie realizată la sfârşitul procedurii).
- Salt la prima instrucţiune a procedurii.

Am putea concluziona că instrucţiunea CALL *adr* este echivalentă cu două instrucţiuni succesive (integrate într-una singură):

```
PUSH PC      ;salvare adresă de revenire în stivă
JUMP adr     ;salt la prima instrucţiune din procedură
```

2.2.2 Formatul instrucţiunilor din clasa ID

Clasa ID conţine instrucţiuni fără operanzi şi instrucţiuni cu operanzi impliciţi (referinţa la operand nu este explicită, prin câmpurile MA şi ADR, ci implicită, prin câmpul OPCODE). Fiind instrucţiuni fără referinţă explicită la operand înseamnă că, în formatul instrucţiunii, nu mai sunt necesare câmpurile MA şi respectiv ADR. Prin urmare, în formatul instrucţiunilor din clasa ID, va apărea doar câmpul OPCODE, extins la 16 biţi (figura 2.6).

Fig 2.6 Formatul instrucţiunilor din clasa ID

Pentru că vom defini mai puţin de 2^5=32 instrucţiuni în cadrul clasei ID şi pentru că urmărim simplificarea decodificatorului instrucţiunii ce urmează a fi implementat în procesorul nostru, vom utiliza drept opcode propriu-zis tot biţii $IR_{14\div10}$; Singura deosebire care va rămâne, în raport cu clasa IOP, este că bitul IR_{15} va avea valoarea 1 specificând astfel că este vorba de o instrucţiune din clasa ID şi nu IOP. Codificarea instrucţiunilor din clasa ID este redată în tabelul 2.3.

Pentru codificarea instrucţiunilor din clasa ID am utilizat doar un subcâmp ($IR_{15}\div IR_{10}$) dintr-un OPCODE pe 16 biţi ($IR_{15}\div IR_0$). Câmpul neutilizat ($IR_9\div IR_0$) va putea fi utilizat în viitor dacă se va dori extensia setului de instrucţiuni. Pentru setul actual putem decide că biţii neutilizaţi ($IR_9\div IR_0$) trebuie să fie zero pentru instrucţiunile din clasa ID (definite în tabelul 2.3); unul sau mai mulţi biţi în unu în câmpul $IR_9\div IR_0$ înseamnă cod ilegal. O astfel de situaţie este semnalizată prin CIL=1 pe ieşirea porţii A din figura 2.8. În acest fel procesorul va recunoaşte ca instrucţiuni legale doar instrucţiunile definite în tabelele 2.2 şi 2.3; orice altă combinaţie în câmpul OPCODE, atât pentru clasa IOP cât şi pentru clasa ID, înseamnă cod ilegal (instrucţiune ilegală).

Primele patrusprezece instrucţiuni definite în clasa ID (INC A, DEC A, ..., RRC A) sunt instrucţiuni care fac referinţă la un operand dar acesta este referit implicit; doisprezece dintre cele patrusprezece instrucţiuni referă registrul A iar două referă registrul FLAG. Spre deosebire de regiştrii generali ($R_{15}\div R_0$) care sunt adresaţi cu cei patru biţi ai câmpului ADR.REG. din codul instrucţiunii (vezi figura 2.5), registrii A şi respectiv FLAG sunt registrii cu funcţii speciale şi prin urmare nu pot fi referiţi prin câmpurile MA şi respectiv ADR. Referinţa la A şi respectiv FLAG nu se poate face decât implicit, prin OPCODE.

Ultimele şase instrucţiuni din tabelul 2.3 (NOP, HALT, EI, DI, RET, RETI) sunt instrucţiuni fără operanzi.

Instrucţiunile ASL A (*Arithmetic Shift Left*) şi respectiv ASR A (*Arithmetic Shift Right*) sunt instrucţiuni de deplasare aritmetică şi acţionează asupra registrului acumulator. Conţinutul acumulatorului este deplasat cu o poziţie, în sensul specificat de instrucţiune. Aceste deplasări operează cu date aritmetice reprezentate în cod complementar (complement faţă de 2) şi prin urmare realizează operaţii aritmetice asupra registrului A.

OPCODE						INSTRUCŢIUNE	
IR_{15}	IR_{14}	IR_{13}	IR_{12}	IR_{11}	IR_{10}		
1	0	0	0	0	0	INC A	;A←A+1 (incrementare acumulator)
1	0	0	0	0	1	DEC A	;A←A-1 (decrementare acumulator)
1	0	0	0	1	0	NEG A	;A← $\overline{A}$ (complementare „bit cu bit” acumulator)
1	0	0	0	1	1	PUSH A	;STIVĂ←A (salvare acumulator în stivă)
1	0	0	1	0	0	PUSH FLAG	;STIVĂ←FLAG (salvare reg. FLAG în stivă)
1	0	0	1	0	1	POP A	;A←STIVĂ (restaurare acumulator din stivă)
1	0	0	1	1	0	POP FLAG	;FLAG←STIVĂ (restaurare FLAG din stivă)
1	0	0	1	1	1	ASL A	;A←A deplasat aritmetic la stânga cu o poziţie
1	0	1	0	0	0	ASR A	;A←A deplasat aritmetic la dreapta cu o poziţie
1	0	1	0	0	1	LSR A	;A←A deplasat logic la dreapta cu o poziţie
1	0	1	0	1	0	ROL A	;A←A rotit la stânga cu o poziţie
1	0	1	0	1	1	ROR A	;A←A rotit la dreapta cu o poziţie
1	0	1	1	0	0	RLC A	;(A,*Carry*)←(A,*Carry*) rotit la stânga cu o poz.
1	0	1	1	0	1	RRC A	;(A,*Carry*)←(A,*Carry*) rotit la dreapta cu o poz.
1	0	1	1	1	0	NOP	;nici o operaţie (*No OPeration*)
1	0	1	1	1	1	HALT	;oprire
1	1	0	0	0	0	EI	;validare întreruperi (*Enable Interrupts*)
1	1	0	0	0	1	DI	;invalidare întreruperi (*Disable Interrupts*)
1	1	0	0	1	0	RET	;Revenire din procedură (*RETurn*)
1	1	0	0	1	1	IRET	;Revenire din *handler* (*Interrupt RETurn*)
1	1	0	1	0	0	—	
1	1	0	1	0	1	—	
1	1	0	1	1	0	—	
1	1	0	1	1	1	—	;Combinaţii neutilizate. Rămân disponibile
1	1	1	0	0	0	—	;pentru dezvoltări viitoare (la nevoie pot fi
1	1	1	0	0	1	—	;definite noi instrucţiuni în clasa ID)
1	1	1	0	1	0	—	
1	1	1	0	1	1	—	
1	1	1	1	0	0	—	
1	1	1	1	0	1	—	
1	1	1	1	1	0	—	
1	1	1	1	1	1	—	

Tabelul 2.3 Codificarea instrucţiunilor din clasa ID

Instrucţiunea ASL A realizează de fapt o înmulţire cu 2 (A←A×2) iar instrucţiunea ASR A realizează o împărţire la 2 (A←A/2). Spre deosebire de instrucţiunile de înmulţire şi respectiv împărţire care au o execuţie complexă (adunări + deplasări repetate) şi care n-au fost prevăzute în setul de instrucţiuni aferent procesorului didactic deoarece am considerat că exced obiectivului „set minimal de instrucţiuni”, instrucţiunile ASL A şi respectiv ASR A au o execuţie foarte rapidă (se execută practic într-o singură perioadă de tact procesor). Este motivul pentru care se recomandă utilizarea acestor instrucţiuni ori de câte ori se doreşte o înmulţire, respectiv împărţire, cu/la 2.

Instrucţiunea LSR A (*Logical Shift Right*) realizează o deplasare logică la dreapta a conţinutului registrului A (interpretează conţinutul acumulatorului ca pe un simplu cuvânt binar de 16 biţi şi nu ca pe un număr reprezentat în cod complementar, cu bit de semn şi respectiv parte de mărime).

O instrucţiune LSL A (*Logical Shift Left*) ar fi identică (din punctul de vedere al execuţiei) cu instrucţiunea ASL A. Din acest motiv n-a fost prevăzută în setul de instrucţiuni; ea va exista ca mnemonică la nivelul limbajului de asamblare dar compilatorul (asamblorul) va genera acelaşi cod maşină atât pentru LSL A cât şi pentru ASL A.

Instrucţiunile ROL A (*ROtate Left*) şi ROR A (*ROtate Right*) realizează rotirea cu o poziţie, în sensul specificat, a conţinutului registrului A. Aceste instrucţiuni realizează de fapt deplasări logice stânga/dreapta pe buclă închisă (bitul care iese dintr-un capăt al registrului A este reintrodus în acelaşi registru pe la celălalt capăt).

Instrucţiunile RLC A (*Rotate Left through Carry*) şi RRC A (*Rotate Right through Carry*) relizează rotiri pe 17 biţi (introduc şi *flag*-ul C în circuitul de rotire).

Instrucţiunea NOP (*No OPeration*) nu face nici o operaţie. Este utilă la construcţia rutinelor de întârziere (*delay*). Este utilă de asemenea compilatoarelor cu reorganizare la implementarea unor strategii statice de reducere a timpului global de execuţie a unui program prin reordonarea instructiunilor din cadrul acestuia. Este utilă şi la implementarea unor strategii statice de predicţie a instrucţiunilor de salt condiţionat.

Instrucţiunile HALT, EI şi DI sunt instrucţiuni de control. Procesoarele moderne execută aceste instrucţiuni numai în mod *supervisor* (modul privilegiat sub care rulează componentele sistemului de operare). În mod *user* (neprivilegiat) aceste instrucţiuni sunt convertite automat în NOP. Toate programele de aplicaţii sunt executate în mod *user*.

Instrucţiunea HALT setează un bistabil care blochează tactul procesorului. Deblocarea se poate realiza numai activând intrarea *RESET* a (pinul *RESET* al) procesorului (prin acţionarea butonului de *RESET* al calculatorului), prin care se va iniţializa procesorul şi de fapt întregul sistem. După iniţializare sistemul trece la încărcarea în memoria internă a sistemului de operare (ca la punerea sub tensiune). Este evident că la o instrucţiune HALT nu poate conduce decât sistemul de operare şi numai în cazul detectării unei erori majore.

Instrucţiunea EI (*Enable Interrupts*) validează întreruperile externe în timp ce instrucţiunea DI (*Disable Interrupts*) le invalidează. Prin invalidare procesorul devine insensibil la cererile de întrerupere externe. O secvenţă de instrucţiuni neinteruptibilă începe cu o instrucţiune DI şi se încheie cu o instrucţiune EI. Pe durata secvenţei neinteruptibile întreruperile externe sunt mascate (invalidate). Doar după execuţia instrucţiunii EI de la sfârşitul secvenţei respective, vor fi luate în considerare şi tratate, într-o ordine prioritară, eventualele cereri de întrerupere active.

Instrucţiunea RET (*RETurn*) va fi intotdeauna ultima instrucţiune dintr-o procedură. Procedurile sunt apelate cu instrucţiuni CALL. Instrucţiunea RET de la sfârşitul procedurii realizează revenirea în programul apelant.

Instrucţiunea IRET (*Interrupt RETurn*) va fi întotdeauna ultima instrucţiune dintr-un *handler* de întrerupere. *Handler*-ele de întrerupere sunt programe din cadrul sistemului de operare. Fiecare periferic are un *handler* propriu (*handler* de disc, *handler* de imprimantă, *handler* de tastatură etc.) prin care sistemul face, perifericului, serviciile solicitate. Serviciile sunt deci implementate în *software* (în sistemul de operare) dar sunt declanşate de *hardware* (cereri de întrerupere). La activarea unei cereri de întrerupere se întrerupe *task*-ul în curs şi este lansat în execuţie *handler*-ul (un nou *task*). *Handler*-ele se încheie cu instrucţiunea IRET care face revenirea în *task*-ul întrerupt. Acesta va fi reluat din punctul din care a fost întrerupt şi continuat ca şi cum n-ar fi fost întrerupt.

În figura 2.7 este sintetizată codificarea instrucţiunilor procesorului didactic. Cu IR_{15}=0 sunt codificate instrucţiunile din clasa IOP (cele 26 instrucţiuni din tabelul 2.2 la care se adaugă cele 6 coduri nealocate). Prin urmare, clasa IOP ar putea fi extinsă (la nevoie) cu maximum 6 instrucţiuni noi. Cu IR_{15}=1 sunt codificate instrucţiunile din clasa ID (cele 20 instrucţiuni din trabelul 2.3 la care se adaugă cele 12 coduri nealocate). Având însă la dispoziţie un OPCODE extins la 16 biţi (figura 2.6), clasa ID ar putea fi extinsă cu un număr foarte mare de instrucţiuni noi.

Toate codurile nealocate trebuie considerate coduri (instrucţiuni) ilegale şi tratate ca atare.

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		
0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	LD A, src	Clasa IOP
0	0	0	0	0	1	X	X	X	X	X	X	X	X	X	X	LD SP, src	
0	0	0	0	1	0	X	X	X	X	X	X	X	X	X	X	LD FLAG, src	
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	⋮	
0	1	1	0	0	1	X	X	X	X	X	X	X	X	X	X	CALL adr	
0	1	1	0	1	0	X	X	X	X	X	X	X	X	X	X	Cod nealocat	CIL
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	⋮	
0	1	1	1	1	1	X	X	X	X	X	X	X	X	X	X	Cod nealocat	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	INC A	Clasa ID
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	DEC A	
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	NEG A	
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	⋮	
1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	IRET	
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Cod nealocat	CIL
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	⋮	
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	Cod nealocat	

Fig 2.7 Codificarea instrucţiunilor procesorului didactic

Utilizarea aceluiaşi câmp de OPCODE propriu-zis (IR_{14}÷IR_{10}) pentru ambele clase de instrucţiuni şi respectiv a bitului IR_{15} pe post de selector de clasă conduce la o implementare extrem de simplă a decodificatorului instrucţiunii (figura 2.8).

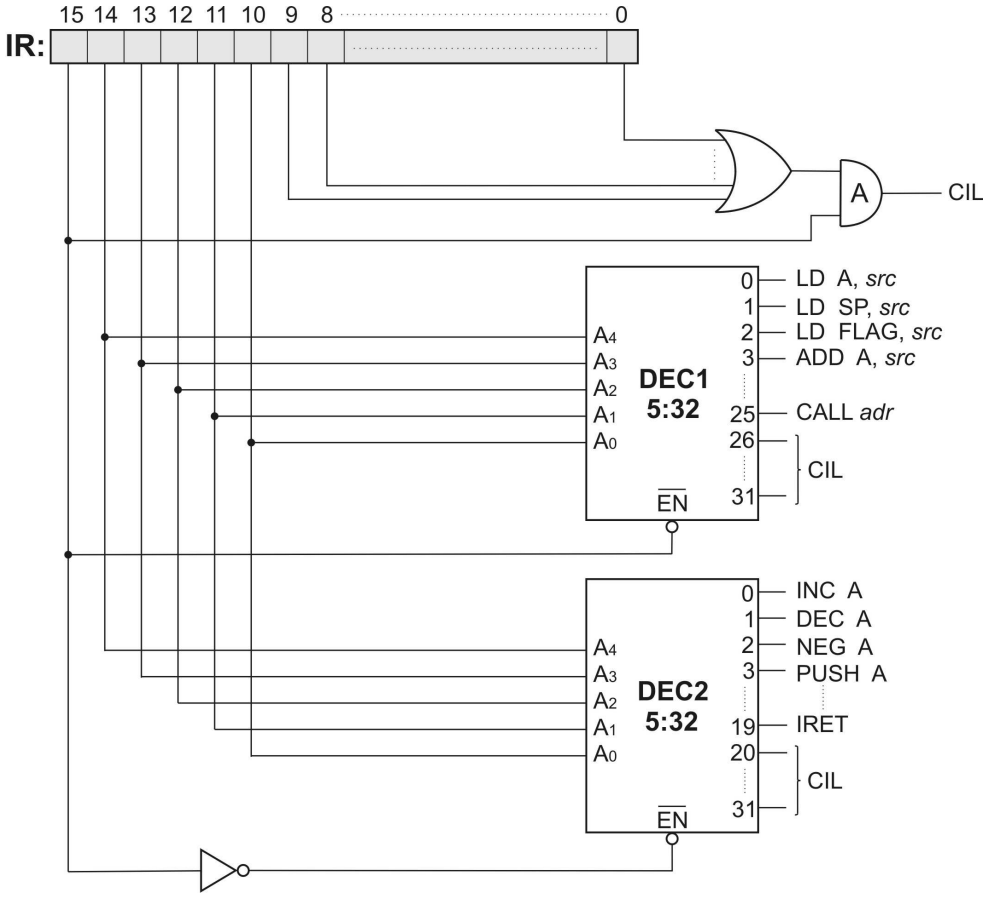


Fig 2.8 Decodificatorul instrucţiunii

Decodificatorul DEC_1 decodifică instrucţiunile din clasa IOP; este validat cu IR_{15}=0, adică dacă în registrul IR se află o instrucţiune IOP. Decodificatorul DEC_2 decodifică instrucţiunile din clasa ID (validat cu IR_{15}=1). În figura 2.7 se observă că instrucţiunile din clasa ID sunt codificate cu biţi de zero pe zona inferioară de *OPCODE* (IR_9=IR_8=IR_7=IR_6=IR_5=IR_4=IR_3=IR_2=IR_1=IR_0=0); unul sau mai mulţi biţi în 1 în această zonă conduce la condiţia cod ilegal (*illegal instruction*). O astfel de situaţie este semnalată de poarta A din figura 2.8. De asemenea, atât pentru clasa IOP cât şi pentru clasa ID, există chiar şi în zona de *OPCODE* propriu-zis (IR_{14}÷IR_{10}) combinaţii nealocate (ilegale). Aceste

combinaţii ilegale evidenţiate în figura 2.7, vor activa condiţia CIL pe ieşirile corespondente aferente decodificatoarelor DEC_1 şi respectiv DEC_2 din figura 2.8. Condiţia CIL globală se va obţine printr-o funcţie SAU aplicată tuturor ieşirilor notate CIL în figura 2.8.

Se poate observa că există rezerve substanţiale pentru extinderea setului de instrucţiuni. O instrucţiune nouă poate fi definită prin transformarea unui cod ilegal în cod legal (prin alocarea codului respectiv noii instrucţiuni).

2.3 Ciclul instrucţiunii

Durata de timp în care o instrucţiune este procesată complet se numeşte ciclul instrucţiunii. Ciclul instrucţiunii este compus din subcicluri pe care le vom numi faze. În timpul procesării ei, o instrucţiune parcurge succesiv următoarele faze:

2.3.1 Faza *Fetch* Instrucţiune (IF)

Faza *fetch* instrucţiune, pentru care vom utiliza abrevierea IF (*Instruction Fetch*), este prima fază din ciclul instrucţiunii. În faza IF instrucţiunea este extrasă din memorie şi încărcată în procesor pentru decodificare şi apoi execuţie.

Instrucţiunile unui program sunt procesate secvenţial, una câte una. Registrul PC conţine în permanenţă adresa instrucţiunii ce urmează a fi procesată. Prin urmare, în faza IF se iniţiază o citire din memorie de la adresa furnizată de PC. Se extrage codul instrucţiunii din locaţia de memorie astfel selectată şi se încarcă în procesor, în registrul IR. După extragerea instrucţiunii, registrul PC este incrementat (pregătit anticipat pentru *fetch*-ul următoarei instrucţiuni).
În concluzie, în faza IF se execută următoarele operaţii elementare:

- se citeşte locaţia de memorie adresată de PC (codul instrucţiunii) şi se încarcă în IR
- se incrementează PC-ul (se pregăteşte anticipat pentru *fetch*-ul următoarei instrucţiuni).
- imediat după incărcarea în IR decodificatorul instrucţiunii decodifică codul încărcat. Codul rămâne în IR până la următoarea încărcare (*fetch*-ul următoarei instrucţiuni).

IF este prima fază din ciclul instrucţiunii şi are un conţinut standard (acelaşi pentru toate instrucţiunile).

2.3.2 Faza *Fetch* Operand (OF)

Imediat după incărcarea instrucţiunii în IR, decodificatorul instrucţiunii decodifică codul încărcat. În faza OF (*Operand Fetch*) se localizează operandul (în acord cu modul de adresare codificat în instrucţiune), se citeşte (din memorie dacă este cazul) şi se încarcă în registrul de manevră M.

2.3.3 Faza de Execuţie (EX)

Faza EX (*Execution*) este specifică fiecărei instrucţiuni. În cadrul ei se execută operaţia specificată de instrucţiune.

Exemple:

- transferul operandului din M în A, în cazul instrucţiunii LD A,src
- adunarea celor doi operanzi (A+M) şi depunerea rezultatului în A, în cazul instrucţiunii ADD A,src
- calculul adresei de salt şi încărcarea acesteia în PC, în cazul instrucţiunii JMP adr
- salvarea registrului PC în stivă, în cazul înstrucţiunii PUSH PC

O dată cu sfârşitul fazei EX se finalizează practic şi procesarea instrucţiunii (se încheie ciclul instrucţiunii). Dacă linia de întrerupere este inactivă, de aici se revine la faza IF, pentru *fetch*-ul următoarei instrucţiuni (din cadrul programului aflat în execuţie).

2.3.4 Faza de Întrerupere (INT)

După cum am afirmat mai sus, sfârşitul fazei EX coincide cu sfârşitul instrucţiunii. Aici se introduce punctul interuptibil, în care se testează cererea globală de întrerupere INTR (*INTerrupt Request*). Dacă INTR=0 (nu există întrerupere externă activă), din EX se trece în IF pentru *fetch*-ul următoarei instrucţiuni din program. Dacă INTR=1 (există o întrerupere externă activă), din EX se trece în faza de întrerupere INT (*Interrupt*). Faza INT suspendă programul în curs şi transferă controlul la o rutină de tratare a întreruperii, denumită, în terminologia de specialitate, *handler* de întrerupere. *Handler*-ele sunt programe din componenţa sistemului de operare. Fiecare periferic are un *handler* propriu (*handler* de tastatură, *handler* de imprimantă, *handler* de disc, *handler* de *mouse* etc) prin care sistemul de operare face servicii perifericului respectiv.

Exemple:

- Tastatura activează o întrerupere când utilizatorul apasă o tastă. Programul în curs, oricare ar fi el, este întrerupt şi se lansează în execuţie *handler*-ul de tastatură (alt program). Acesta citeşte codul tastei apăsate din portul de date al tastaturii (încărcarea codului în port se face sub controlul *hardware*-ului) şi-l încarcă în buferul de tastatură (o tabelă a sistemului de operare, rezidentă în memorie, unde se „asamblează” comenzile date de la tastatură). Ultima instrucţiune a *handler*-ului este IRET, prin care se revine în programul întrerupt (suspendat temporar). Acesta va fi reluat cu următoarea instrucţiune şi continuat ca şi cum n-ar fi fost întrerupt. La apăsarea următoarei taste lucrurile decurg analog.

- Imprimanta activează o întrerupere atunci când, în procesul de tipărire, goleşte buferul de caractere. Prin întrerupere solicită încărcarea în bufer a următoarelor

caractere de tipărit. Această încărcare cade în sarcina *handler*-ului de imprimantă. O nouă întrerupere va fi activată la următoarea golire a *bufer*-ului.

- *Mouse*-ul activează o înterrupere atunci când utilizatorul acţionează unul din butoanele sale. *Handler*-ul de *mouse* va executa (pe ecran) funcţia aferentă butonului respectiv. La următoarea acţionare a unui buton de *mouse* acelaşi *handler* este lansat din nou în execuţie.

Prin urmare, faza INT are rolul de a întrerupe programul în curs şi de a lansa în execuţie un alt program numit *handler* de întrerupere. Întreruperea trebuie astfel făcută încât programul întrerupt să sufere doar o suspendare temporară (să poată fi reluat prin instrucţiunea IRET de la sfârşitul *handler*-ului). Principiul procesării întreruperilor este redat în figura 2.9.

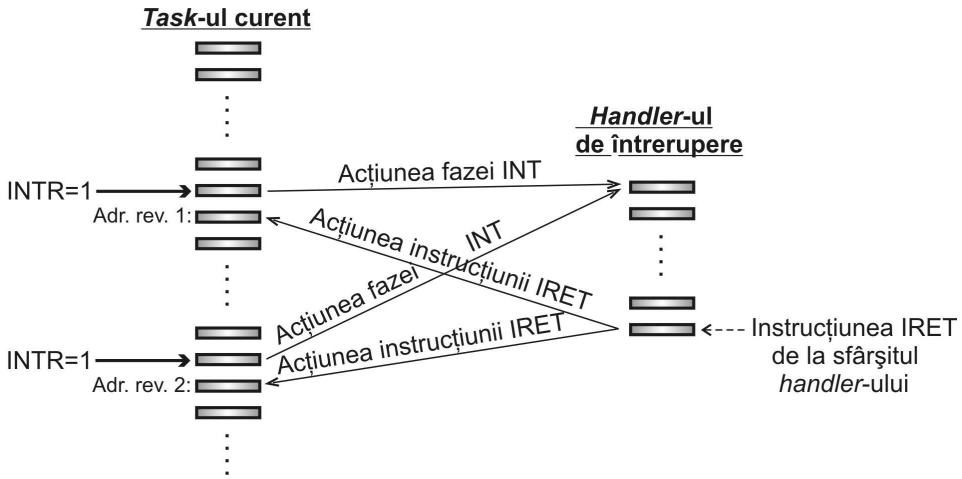


Fig 2.9 Principiul procesării întreruperilor

Principial, faza INT trebuie să rezolve următoarele probleme:

1. Să facă posibilă reluarea programului întrerupt (suspendat temporar), prin instrucţiunea IRET de la sârşitul *handler*-ului; toate *handler*-ele se încheie cu instrucţiunea IRET.

2. Să conserve starea *flag*-urilor de condiţii din momentul întreruperii astfel încât instrucţiunea IRET să refacă această stare în *flag*-uri. Reluarea *task*-ului întrerupt trebuie făcută cu *flag*-urile de condiţii exact în starea pe care o aveau în momentul întreruperii.

3. Să „afle” adreasa de start a *handler*-ului, s-o încarce în registrul PC şi să determine unitatea de control să treacă procesorul în faza IF. Se va face astfel *fetch*-ul primei instrucţiuni din *handler* (se lansează în execuţie *handler*-ul de întrerupere).

Pentru rezolvarea celor trei probleme enumerate se aplică următoarele soluţii:

1. În faza INT se salvează în stivă registrul PC. În momentul întreruperii registrul PC conţine tocmai adresa de revenire în *task*-ul ce urmează a fi întrerupt. Instrucţiunea IRET de la sfârşitul *handler*-ului va trebui să restaureze din stivă registrul PC.

2. În faza INT se salvează în stivă registrul FLAG. Instrucţiunea IRET de la sfârşitul *handler*-ului va trebui să restaureze din stivă registrul FLAG. Astfel revenirea în *task*-ul temporar suspendat, se face cu *flag*-urile exact în starea pe care o aveau în momentul întreruperii.

3. În faza INT se achită cererea de întrerupere INTR prin activarea unui semnal de răspuns INTA (*INTerrupt Acknowledge*). Răspunsul INTA ajunge la perifericul care a solicitat întreruperea (la *controller*-ul I/O aferent perifericului), determinându-l să-şi dezactiveze cererea (serviciul a fost iniţiat şi prin urmare cererea de întrerupere trebuie dezactivată). Sistemul de întreruperi şi excepţii (SIE) va genera vectorul de întrerupere (VI) aferent perifericului care a solicitat întreruperea şi va încăca acest vector în registrul IVR (*Interrupt Vector Register*). VI este o adresă de memorie (un pointer care selectează o locaţie din tabela vectorilor de întrerupere). Această tabelă, denumită în terminologia de spacialitate IVT (*Interrupt Vector Table*), este o tabelă a sistemului de operare, rezidentă în RAM, care conţine adresele de start ale *handler*-elor de întrerupere aferente tuturor perifericelor existente în sistem. Fiecere periferic are un vector de întrerupere propriu, care pointează în tabelă pe o locaţie iar locaţia conţine adresa de start a *handler*-ului propriu. De aici rezultă că nu este permisă alocarea aceluiaşi vector de întrerupere la două periferice diferite. În consecinţă, memoria trebuie adresată cu VI, trebuie citit conţinutul locaţiei accesate (adresa de start a *handler*-ului din locaţia selectată în IVT) şi acest conţinut trebuie încărcat în PC. Trebuie să se treacă apoi în faza IF; se va face *fetch*-ul primei instrucţiuni din *handler*.

Handler-ul va fi executat ca orice program. Toate *handler*-ele se încheie cu instrucţiunea IRET care va restaura succesiv doi regiştrii din stivă: PC şi FLAG. Să notăm că trebuie respectat principiul LIFO (*Last-In First-Out*) pe care se bazează stiva. Prin urmare, dacă decidem că ordinea celor două *pop*-uri succesive din cadrul fazei de execuţie aferentă instrucţiunii IRET va fi *pop* PC urmat de *pop* FLAG, atunci *push*-urile din faza INT trebuie făcute în ordine inversă: *push* FLAG urmat de *push* PC. Să mai notăm că:

1. Atâta timp cât întreruperile sunt validate, *handler*-ele de întrerupere pot fi, la rândul lor, întrerupte (figura 2.10). Este valid aşa zisul mecanism al întreruperilor cascadate („întrerupere în întrerupere”) dar trebuie precizat că un *handler* de întrerupere nu poate fi întrerupt decât de o întrerupere cu prioritate strict mai mare decât prioritatea celei care l-a lansat în execuţie. Practic, fiecărui periferic i se alocă o linie proprie de întrerupere (IRQ_i) şi aceste linii de întrerupere sunt prioritizate. Asta înseamnă că, suma dintre numărul de linii de întrerupere (periferice) şi respectiv numărul de excepţii reprezintă lungimea maximă pe care o poate atinge laţul de întreruperi cascadate. Întreruperile şi excepţiile sunt însă evenimente foarte rare în raport cu viteza (rata) de procesare a instrucţiunilor în procesor. Prin urmare, statistic, lanţul de întreruperi cascadate va avea o lungime foarte mică.

2. Necesitatea salvării-restaurării registrului FLAG în cadrul mecanismului INT-IRET este ilustrată în figura 2.11. În raport cu procesarea instrucţiunilor în procesor, întreruperile au un caracter asincron şi prin urmare, există posibilitatea ca o întrerupere să se activeze în timpul procesării unei instrucţiuni aritmetico-logice plasată imediat înaintea unei instrucţiuni de salt condiţionat (în figura 2.11 instrucţiunea CMP A,*src* plasată imediat înaintea instrucţiunii JZ *adr*). Pentru verificarea condiţiei de salt, instrucţiunea JZ *adr* testează de fapt *flag*-ul Z poziţionat de instrucţiunea CMP A,*src*. Dar şi *hanler*-ul poate conţine instrucţiuni aritmetice care poziţionează *flag*-urile de condiţii (instrucţiunea ADD A,*src* din figura 2.11). Dacă, în cadrul mecanismului INT-IRET, nu s-ar salva şi respectiv restaura şi registrul FLAG, atunci instrucţiunea JZ *adr* ar testa *flag*-ul Z poziţionat de ultima instrucţiune aritmetică din *handler* (în exemplul nostru, instrucţiunea ADD A,*src*). Evident că programul reluat riscă să fie continuat incorect. Salvarea în stivă a registrului FLAG în faza INT, urmată de restaurarea din stivă a aceluiaşi registru în faza EX a instrucţiunii IRET, rezolvă această problemă.

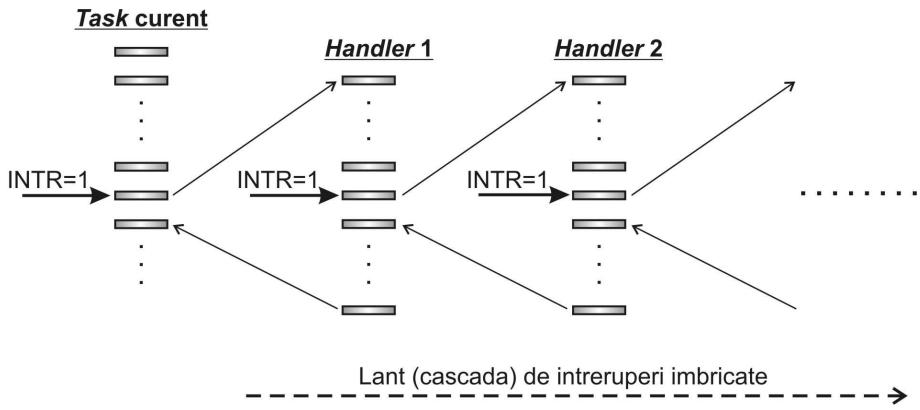


Fig. 2.10 Principiul cascadării întreruperilor

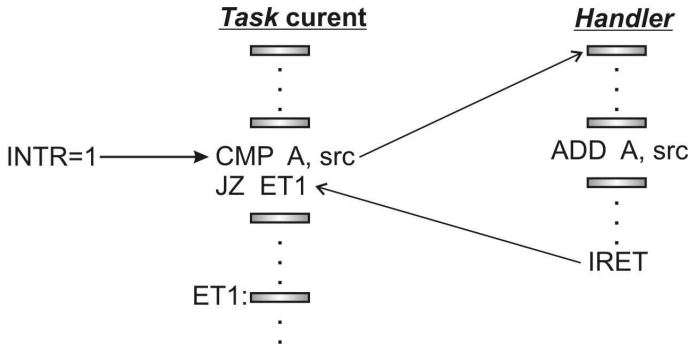


Fig 2.11 Necesitatea salvării/restaurării registrului FLAG
în cadrul mecanismului INT–IRET

Ţinând cont de cele prezentate putem rezuma operaţiile succesive care vor fi executate în cadrul fazei INT:

- se achită întreruperea (la cererea INTR procesorul răspunde cu INTA) pentru a determina perifericul (*controller*-ul I/O aferent) să dezactiveze cererea şi respectiv SIE să genereze şi să încarce vectorul de întrerupere în registrul IVR.
- se salvează în stivă registrul FLAG (starea *flag*-urilor din momentul întreruperii).
- se salvează în stivă registrul PC (adresa de revenire în *task*-ul întrerupt).
- se citeşte locaţia de memorie adresată de VI (se citeşte adresa de start a *handler*-ului din tabela IVT) şi se încarcă în PC.

După faza INT urmează întotdeauna faza IF (*fetch*-ul primei instrucţiuni din *handler*). *Handler*-ul poate fi, la rândul lui întrerupt, de o întrerupere de prioritate mai mare. Există aşa zise secţiuni critice de cod care trebuie executate cu întreruperile invalidate. *Handler*-ele sunt *task*-uri din componenţa sistemului de operare, situate pe cel mai de jos nivel (interacţionează intim cu *hardware*-ul) şi din acest motiv conţin de regulă astfel de secţiuni critice. La intrarea într-o secţiune critică, întreruperile trebuie să fie invalidate cu ajutorul instrucţiunii DI (*Disable Interrupts*). La sfârşitul secţiunii critice va apărea instrucţiunea EI (*Enable Interrupts*). După execuţia instrucţiunii EI, *handler*-ul devine (din nou) interuptibil şi dacă există o eventuală întrerupere activă în aşteptare, de prioritate mai mare decât prioritatea celei care a lansat în execuţie *handler*-ul curent, în acest moment va fi deservită (întreruperi cascadate).

2.4 Succesiunea fazelor

Succesiunea fazelor în timpul procesării instrucţiunilor este redată în figura 2.12.
Ciclul instrucţiunii conţine:

- trei faze (IF+OF+EX) în cazul instrucţiunilor din clasa IOP
- două faze (IF+EX) în cazul instrucţiunilor din clasa ID

În faza OF se face *fetch*-ul operandului (referit explicit prin câmpurile MA şi respectiv ADR din codul instrucţiunii). Instrucţiunile din clasa ID sunt instrucţiuni fără operanzi (sau cu operanzi referiţi implicit prin câmpul OPCODE). Din acest motiv, pentru instrucţiunile ID, se evită faza OF (din IF se trece direct în EX).

Ciclul instrucţiunii se încheie la sfârşitul fazei EX. Aici se introduce punctul interuptibil (se testează cererea globală de întrerupere INTR). Din punctul interuptibil:

- se revine în faza IF pentru *fetch*-ul următoarei instrucţiuni din program, dacă INTR=0 (nici-un periferic nu solicită întrerupere).
- se intră în faza de întrerupere INT, dacă INTR=1 (cel puţin un periferic solicită întrerupere).

De la sfârşitul fazei INT se revine la IF, pentru *fetch*-ul primei instrucţiuni din *handler*-ul de întrerupere asociat întreruperii deservite (asociat întreruperii achitate în faza INT).

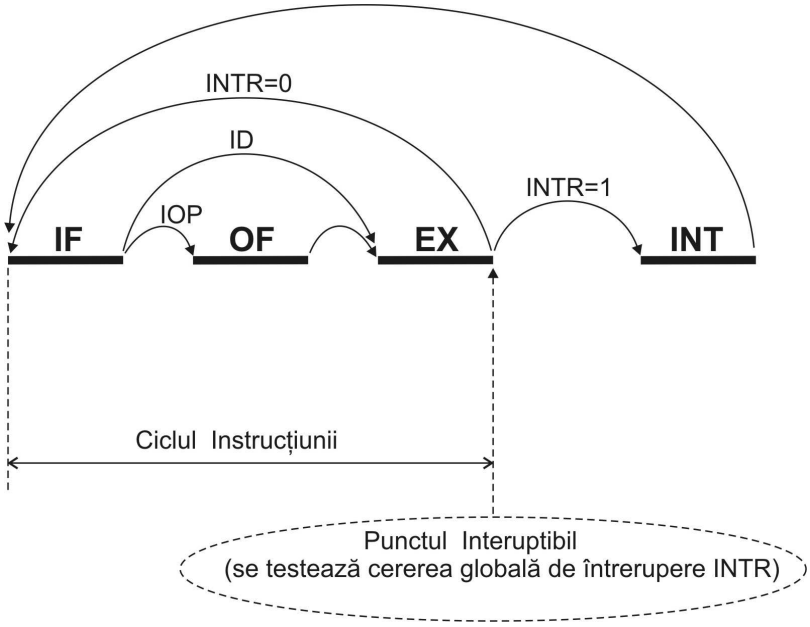


Fig. 2.12 Succesiune fazelor în timpul procesării instrucţiunilor

2.5 Proiectarea fazei IF

Faza *fetch* instrucţiune a fost descrisă principial la 2.3.1. Proiectarea fazei IF înseamnă de fapt detalierea descrierii de la 2.3.1, prin transpunerea acestei descrieri în operaţii elementare succesive. Denumim operaţie elementară acea operaţie care poate fi executată într-o singură perioadă de tact procesor. Platforma *hardware* utilizată pentru proiectarea fazei IF (ca de altfel a tuturor fazelor) va fi procesorul didactic a cărui arhitectură (schemă bloc) a fost prezentată la 2.1 (figura 2.2).
Exemple de operaţii elementare (executabile într-o singură perioadă de tact):

- Transferuri registru–registru. Exemple:

  1. A←M; Pentru execuţia acestui transfer sunt utilizate următoarele resurse *hardware* din structura procesorului didactic: registrul sursă (M), busul DBUS (pe care vor fi transmise datele din M), unitatea ALU (care va transmite pe RBUS, nemodificate, informaţiile preluate de pe DBUS), busul RBUS şi registrul destinaţie (A).

  2. A←A+M; Este un transfer registru–registru cu procesarea în ALU a informaţiilor implicate în transfer. Pentru execuţie sunt utilizate:

registrul sursă 1 (A), registrul sursă 2 (M), busul SBUS (pe care se transferă operandul 1 preluat din A), busul DBUS (pe care se transferă operandul 2 preluat din M), unitatea ALU (care face o operaţie de adunare), busul RBUS (pe care se transferă rezultatul produs de ALU) şi registrul destinaţie (A) în care vor fi încărcate datele prezente pe RBUS (rezultatul ALU).

- Operaţii speciale implementate pe regiştre. Exemple:

  1. PC←PC+2; Această incrementare se va face local, la nivelul registrului PC care va fi dotat şi cu o funcţie de numărare în sus, din 2 în 2, funcţie declanşată la activarea comenzii *+2PC* din figura 2.2.

  2. SP←SP-2; Decrementare cu 2 implementată local la nivelul registrului pointer de stivă (comanda *-2SP* din figura 2.2).

- Operaţii de transfer procesor-memorie (citire din şi respectiv scriere în memorie). Sunt declanşate de comenzile *RD* (citire) şi respectiv *WR* (scriere). Operează asupra locaţiei selectate de adresa prezentă în registrul ADR. Rezultă că, înainte de lansarea unei operaţii de citire din sau de scriere în memorie, în ADR trebuie încărcată adresa locaţiei ce se doreşte a fi accesată. Durata operaţiilor de citire/scriere memorie depinde de timpul de acces la memorie. În cazul memoriilor sincrone rapide (cum ar fi memoria *cache on-chip*) aceste operaţii de citire/scriere memorie se pot încadra într-o perioadă de tact procesor. În cazul memoriilor mai lente cum ar fi memoria centrală (de regulă RAM-dinamic în sistemele actuale), operaţiile de citire/scriere memorie au o durată semnificativ mai mare şi trebuie controlate printr-un protocol asincron bazat pe semnale de *hand-shaking*.

2.5.1 Organigrama fazei IF

Organigrama fazei IF detaliază succesiunea de operaţii elementare din cadrul fazei IF (figura 2.13). Vom explica această organigramă facând apel la numerotarea blocurilor din componenţa acesteia:

(1) • Adresa de *fetch* (adresa instrucţiunii ce urmează a fi extrasă din memorie) se transferă din PC în ADR.

(2) • Se citeşte codul instrucţiunii din memorie (*RD*) şi se încarcă în IR (IR←MEM$|_{ADR}$). Notaţia MEM$|_{ADR}$ semnifică conţinutul locaţiei de memorie selectată de adresa prezentă în ADR (pe scurt, „locaţia selectată de ADR”).

• Se incrementează PC–ul cu 2 (se pregăteşte anticipat pentru adresarea următoarei instrucţiuni din program).

(3),(4),(5) • Dacă ACLOW=1 (dacă a căzut tensiunea alternativă din reţeaua de alimentare a sistemului), se setează bistabilul de excepţie BE_0 şi se trece în faza de întrerupere INT. Setarea bistabilului BE_0 determină generarea

în SIE a vectorului de întrerupere (de excepţie) asociat evenimentului „cădere tensiune de alimentare”. În faza INT vectorul va fi utilizat pentru obţinerea adresei de start a *handler*-ului asociat. *Handler*-ul asociat acestui eveniment, numit şi rutină *POWER FAIL*, va pregăti sistemul de operare pentru oprirea de urgenţă a calculatorului, oprire ce trebuie făcută urgent dar în condiţii de siguranţă. Rutina *POWER FAIL* trebuie executată (procesul de oprire trebuie finalizat) în intervalul de timp dintre evenimentele ACLOW şi respectiv DCLOW (tensiunea continuă de alimentare mai rezistă o perioadă scurtă de timp, după „căderea” celei alternative, pe baza energiei înmagazinate în capacităţile de filtrare din sursa de alimentare).

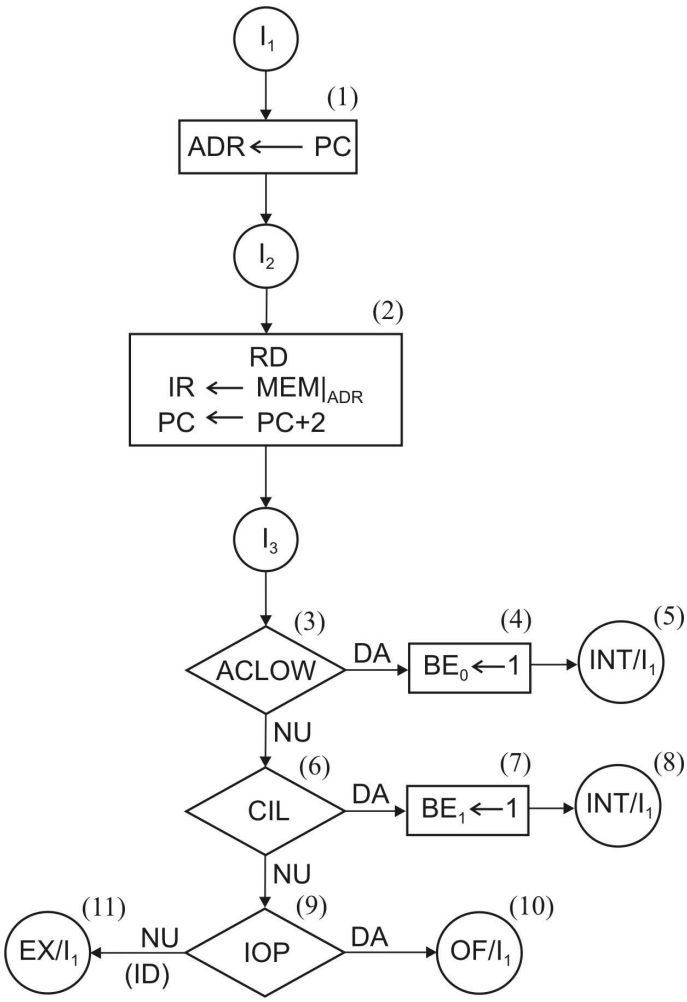


Fig. 2.13 Organigrama fazei IF

(3),(4),(5) • Dacă ACLOW=1 (dacă a căzut tensiunea alternativă din reţeaua de alimentare a sistemului), se setează bistabilul de excepţie BE_0 şi se trece în faza de întrerupere INT. Setarea bistabilului BE_0 determină generarea în SIE a vectorului de întrerupere (de excepţie) asociat evenimentului „cădere tensiune de alimentare”. În faza INT vectorul va fi utilizat pentru obţinerea adresei de start a *handler*-ului asociat. *Handler*-ul asociat acestui eveniment, numit şi rutină *POWER FAIL*, va pregăti sistemul de operare pentru oprirea de urgenţă a calculatorului, oprire ce trebuie făcută urgent dar în condiţii de siguranţă. Rutina *POWER FAIL* trebuie executată (procesul de oprire trebuie finalizat) în intervalul de timp dintre evenimentele ACLOW şi respectiv DCLOW (tensiunea continuă de alimentare mai rezistă o perioadă scurtă de timp, după „căderea” celei alternative, pe baza energiei înmagazinate în capacităţile de filtrare din sursa de alimentare).

(6),(7),(8) • Dacă instrucţiunea încărcată în IR este de fapt un cod ilegal, se setează bistabilul de excepţie BE_1 şi se intră în faza de întrerupere INT. Setarea bistabilului BE_1 determină generarea în SIE a vectorului de întrerupere asociat evenimentului CIL. Faza INT va suspenda *task*-ul în curs şi va lansa în execuţie *handler*-ul asociat excepţiei CIL.

(9),(10),(11) • Dacă instrucţiunea încărcată în IR este din clasa IOP, se trece în faza *fetch* operand (OF) iar dacă instrucţiunea este din clasa ID, se trece în faza de execuţie (EX).

Stările din organigramă ($I_1 \div I_3$) corespund impulsurilor de stare (perioadelor de tact) ce compun faza IF. Pe fiecare perioadă de tact (identificată prin impulsul de stare asociat) se execută una sau mai multe operaţii elementare. Pentru performanţă trebuie urmărit ca fazele să fie cât mai scurte. Prin urmare, ar fi benefică planificarea unui set cât mai amplu de operaţii elementare pe fiecare impuls de stare. Dar, pentru că operaţiile asignate unui impuls sunt de fapt operatii concurente, trebuie evitate conflictele. Nu pot fi executate concurent două operaţii care intră în conflict.

Exemple:

- Nu pot fi emise pe acelaşi bus şi pe durata aceluiaşi impuls de stare conţinuturile a doi regiştrii diferiţi.

- Unitatea ALU nu poate efectua decât o singură operaţie pe durata unui impuls de tact. Încercarea de a planifica două sau mai multe operaţii ALU pe acelaşi impuls de stare conduce la conflicte în ALU.

Pentru simplificare, am considerat că memoria este sincronă rapidă (funcţionează în sincronism cu tactul procesorului şi derulează o operaţie de citire, respectiv scriere, pe durata unui singur impuls de stare). Cu alte cuvinte, am considerat că timpul de acces la memorie este mai mic sau cel mult egal cu perioada de tact procesor ($t_a \leq T_{CLKprocesor}$).

Dacă memoria este mai lentă în raport cu procesorul (cum este memoria RAM-dinamic utilizată actualmente în construcţia memoriilor centrale, care mai are şi cicluri periodice de regenerare perturbatoare) atunci, interfaţa procesor–memorie devine mai complexă (protocol asincron bazat pe semnale de hand-shaking aşa cum indică figura 2.14).

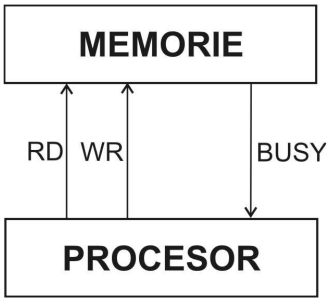


Fig. 2.14 Interfaţa procesor–memorie bazată pe semnale de *hand-shaking*

Protocolul asincron de comunicaţie presupune următorii paşi:

1. Procesorul activează comanda (RD sau WR, funcţie de tipul accesului la memorie).

2. Memoria activează semnalul de răspuns BUSY (devine ocupată cu operaţia solicitată de procesor şi deci indisponibilă pentru alte eventuale accese). Să notăm că există şi alte dispozitive care pot accesa memoria. Un exemplu ar fi *controller*-ul de disc care face accese DMA (*Direct Memory Access*). BUSY rămâne activ până la încheierea ciclului de citire (sau scriere) iniţiat de procesor.

3. Activarea semnalului BUSY va fi sesizată de procesor (BGC) care va bloca tactul intern, impunând astfel o stare de aşteptare.

4. La terminarea operaţiei de ctire din (scriere în) memorie, BUSY este dezactivat.

5. Dezactivarea semnalului BUSY este sesizată de procesor (BGC) care deblochează tactul intern. Dacă operaţia a fost de citire, datele citite din memorie se încarcă în procesor (într-un registru intern) şi apoi se trece în următorul impuls de stare.

Acest protocol poate fi implementat în organigramă, de exemplu pe impulsul de stare I_2, în care se citeşte instrucţiunea din memorie. Ajustarea care s-ar impune este prezentată în figura 2.15.

Pentru simplificare, în continuare vom considera că procesorul didactic este interfaţat cu o memorie sincronă rapidă.

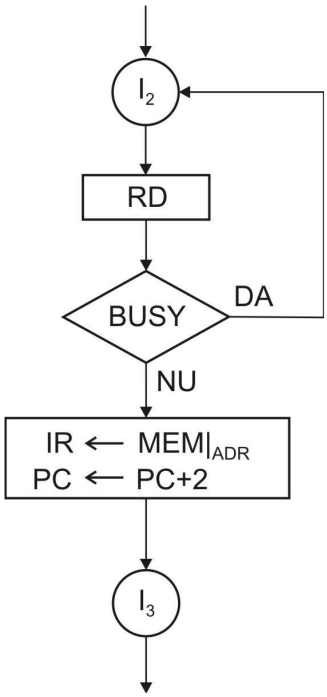


Fig. 2.15 Protocol asincron pentru interfaţa procesor–memorie.
Exemplificare pe operaţia de *fetch* instrucţiune

2.5.2 Tabelul de comenzi aferent fazei IF

Sintetizăm operaţiile elementare din organigrama fazei IF într-un tabel de comenzi (fig 2.16).

Faza IF

I_1	I_2	I_3
PdPC, DBUS, PdALU, PmADR	*RD, PdMEM, PmIR, +2PC*	ACLOW: *A(1)BE0, TINT, TI_1* CIL: *A(1)BE1, TINT, TI_1* IOP: *TOF, TI_1* $\overline{\text{IOP}}$: *TEX, TI_1*

Fig 2.16 Tabelul de comenzi aferent fazei IF

Pentru execuţia unei anumite operaţii vor trebui activate una sau mai multe comenzi elementare (C_L). Coloanele tabelului de comenzi corespund celor trei impulsuri de stare (perioade de tact) din componenţa fazei IF. Fiecare coloană din tabel va conţine comenzile logice elementare ce se vor activa pe impulsul de stare respectiv. Aceste comenzi se deduc

din organigramă. Pentru fiecare operaţie elementară din organigramă, trebuie dedus şi trecut în tabel setul de comenzi elementare aferent. Pentru transpunerea corectă a organigramei în tabel de comenzi trebuie utilizată schema bloc a procesorului didactic (figura 2.2). Pe această schemă bloc, se deduce cu uşurinţă setul de comenzi elementare aferent unei anumite operaţii elementare.

Faza IF se execută în trei impulsuri de stare (figura 2.17). Cele trei impulsuri de stare succesive marchează cele trei perioade de tact din componenţa fazei IF. Pe fiecare coloană din tabelul de comenzi apar evidenţiate toate comenzile logice elementare (C_L) activate pe durata impulsului de stare respectiv (perioadei de tact respective).

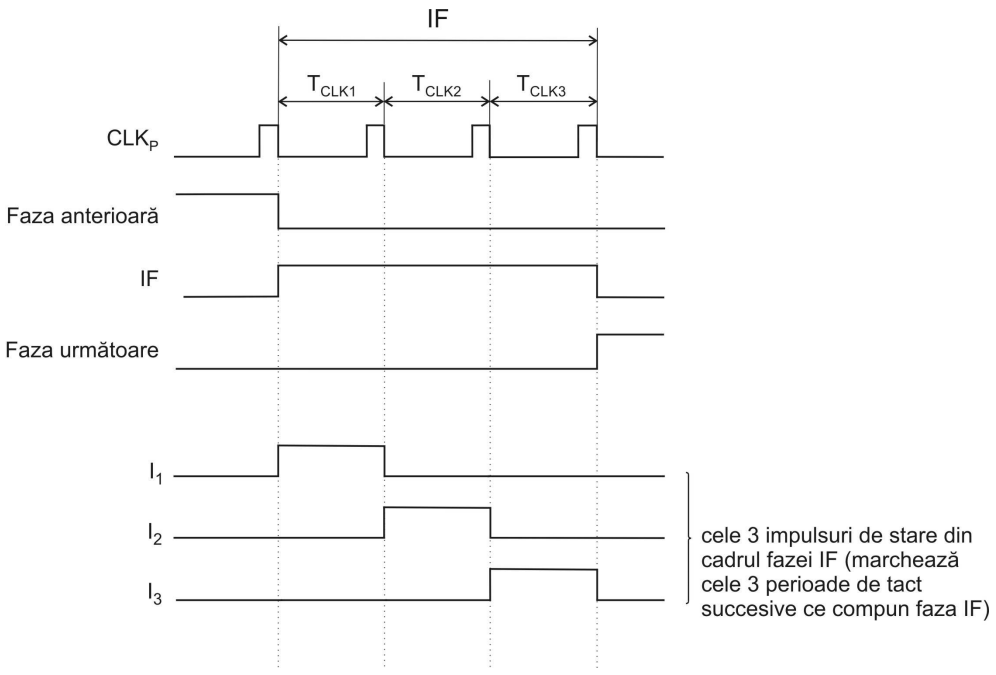


Fig 2.17 Succesiunea de impulsuri de stare din cadrul fazei IF

Cele trei impulsuri de stare succesive marchează cele trei perioade de tact din componenţa fazei IF. Pe fiecare coloană din tabelul de comenzi apar evidenţiate toate comenzile logice elementare (C_L) activate pe durata impulsului de stare respectiv (perioadei de tact respective).

Exemple:

1. Pe I_1 se activează patru comenzi elementare (*PdPC, DBUS, PdALU, PmADR*) care, împreună, vor realiza operaţia: ADR←PC. Cele patru comenzi se deduc pe baza schemei bloc din figura 2.2. În acord cu această schemă bloc, acţiunea fiecărei comenzi poate fi sintetizată astfel:

PdPC –(„predă PC”); emite conţinutul registrului PC pe DBUS.

DBUS –comandă pentru ALU; determină trecerea nemodificată prin ALU a operandului preluat de pe DBUS.

PdALU –(„predă ALU”); emite rezultatul de pe ieşirile ALU pe RBUS.

PmADR –(„primeşte ADR”); încarcă datele prezente pe RBUS în registrul ADR.

Forma acestor comenzi este redată în figura 2.18. Comenzile *PdPC*, *DBUS* şi *PdALU* sunt comenzi de tip nivel (active pe toată durata perioadei de tact) iar comanda *PmADR* este comandă de tip impuls (activă numai pe durata impulsului de tact din cadrul respectivei perioade de tact). Registrul ADR, ca de altfel toţi regiştrii, este format din bistabili (se utilizează de regulă bistabili D). Informaţia transferată se aplică pe intrările D aferente celor 16 bistabili ce compun registrul ADR iar comanda de încărcare (*PmADR*) se aplică pe intrarea C (de *clock*) aferentă celor 16 bistabili. Încărcarea informaţiilor în registru (figura 2.18) se face pe frontul urcător al comenzi *PmADR* (funcţia de încărcare date în bistabilul D). În momentul tranziţiei *low-high* a semnalului de pe intrarea de *clock*, biţii de informaţie trebuie să fie corecţi şi stabili pe intrările D aferente celor 16 bistabili.

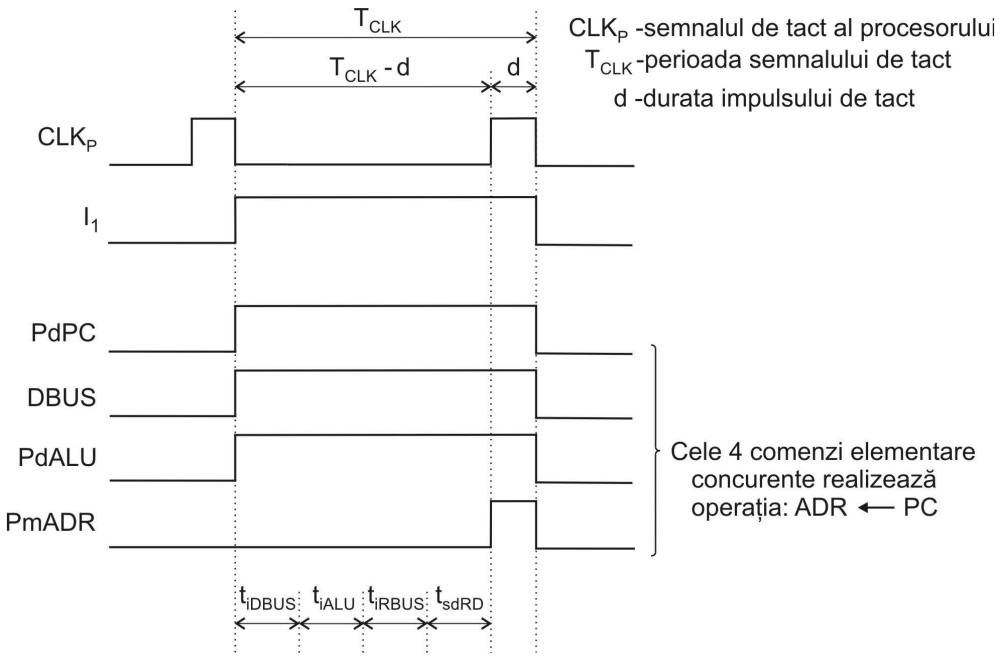


Fig. 2.18 Comenzi de tip nivel şi de tip impuls

Pentru ca această condiţie să fie respectată trebuie satisfăcută relaţia:

$$t_{iDBUS}+t_{iALU}+t_{iRBUS}+t_{sdRD}\leq T_{CLK}-d \qquad (2.1)$$

unde: t_{iDBUS} –timp de întârziere date pe DBUS. Emisia datelor din registrul sursă (PC) pe DBUS se face prin intermediul unor circuite de acces pe DBUS. Având mai multe surse pe DBUS este evident că trebuie implementată o funcţie SAU între aceste surse (eventual un SAU-cablat, utilizânt circuite cu ieşiri *three-state*). Este vorba de timpul de propagare a informaţiei prin aceste circuite.

t_{iALU} –timp de întârziere în ALU (propagarea informaţiei prin ALU).

t_{iRBUS} –timp de întârziere date pe RBUS (este similar timpului de întârziere pe DBUS).

t_{sdRD} –timp de stabilizare date pe intrarea registrului destinaţie (cei 16 biţi de date trebuie să fie prezenţi pe intrările D aferente celor 16 bistabili ce compun registrul ADR, cu t_{sdRD} secunde înaintea frontului urcător de pe intrarea de clock (C).

T_{CLK} –durata perioadei de tact procesor

d –durata impulsului de tact.

2. Pe I_3, dacă CIL=1, se activeaza comenzile *A(1)BE₁*, *TINT*, *TI₁* care realizează următoarele operaţii elementare:

A(1)BE₁ –„adu la 1 BE_1” (setează bistabilul de excepţie BE_1).

TINT –„treci în faza de întrerupere” (comandă destinată generatorului de faze).

TI₁ –„treci în I_1” (comandă de iniţializare, destinată generatorului de impulsuri de stare). Fiecare fază conţine o succesiune de impulsuri de stare (perioade de tact). Succesiunea începe întotdeauna cu I_1 (T_{CLK1}). Din acest motiv, în tabelele de comenzi, o comandă de tipul „treci într-o fază următoare” este acompaniată întotdeauna de comanda *TI₁*.

2.6 Proiectarea fazei OF

După *fetch*-ul instrucţiunii, dacă instrucţiunea referă explicit un operand, urmează *fetch*-ul operandului. Prin urmare, faza *fetch* operand (OF) este parcursă numai dacă instrucţiunea aparţine clasei IOP. În faza OF se localizează operandul specificat explicit în codul instrucţiunii (prin câmpurile MA şi ADR), se citeşte acest operand şi se încarcă în registrul

M. Localizarea operandului se face în funcţie de modul de adresare codificat în câmpul MA din codul instrucţiunii, utilizând schemele de principiu din figura 2.5.

2.6.1 Organigrama fazei OF

Organigrama fazei *fetch* operand este redată în figura 2.19 iar explicaţiile aferente le vom da, apelând la numerotarea blocurilor din componenţa acesteia:

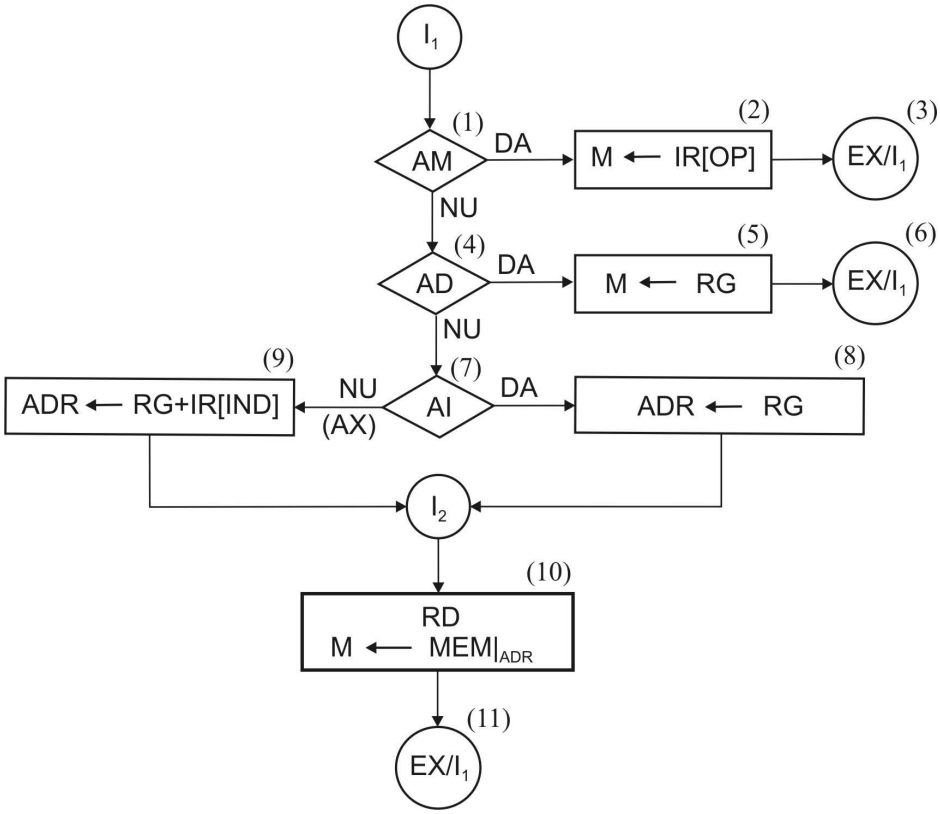


Fig 2.19 Organigrama fazei OF

(1),(2) • În cazul modului de adresare <u>imediat</u> (AM) operandul este chiar octetul inferior din codul instrucţiunii (figura 2.5.a).). Acest operand imediat, abreviat cu notaţia IR[OP] în organigramă, este încărcat în registrul M. Să notăm că operand imediat înseamnă de fapt operand de tip constantă. Exemple de instrucţiuni cu operanzi imediaţi:

```
LD A, 4      ;A←4 (încarcă în acumulator constanta 4). Constanta 4 este
             ;codificată binar în octetul inferior din codul instrucţiunii (vezi
             ;fig 2.5 a).).
```

SUB A, 12 ;A←A-12 (scade constanta 12 din conţinutul acumulatorului).

(4),(5) • În cazul modului de adresare direct (AD) operandul se află în registrul general selectat de câmpul ADR. REG. din codul instrucţiunii (figura 2.5.b).). Este motivul pentru care, modul AD, este denumit uneori (uşor mai riguros) mod de adresare registru-direct (operand în registru). Câmpul ADR. REG. (biţii $IR_{7\div4}$ din codul instrucţiunii) se aplică pe intrările de adresă ale setului de regiştrii generali (matrice de memorare cu 16 locaţii pe 16 biţi) selectând automat registrul specificat în instrucţiune. Rezultă că, atâta timp cât codul instrucţiunii este prezent în IR, registrul general specificat este automat selectat. Va mai fi nevoie de o simplă comandă PdRG („predă registru general”) pentru a emite conţinutul acestui registru pe DBUS (în scopul încărcării lui în registrul M). Exemple de instrucţiuni cu mod de adresare direct:

ADD A, R5 ;A←A+R_5. Conţinutul lui R_5 reprezintă operandul referit cu ;modul de adresare direct.

CMP A, R9 ;A-R_9 pentru poziţionarea *flag*-urilor în acord cu rezultatul. ;Conţinutul lui R_9 reprezintă operandul referit cu modul de ;adresare direct.

(7),(8) • În cazul modului de adresare indirect (AI), registrul general selectat de câmpul ADR. REG. conţine adresa operandului (figura 2.5.c).). Această adresă va fi încărcată în registrul ADR pentru a pregăti aducerea operandului din memorie. Este motivul pentru care, modul AI, este denumit uneori (uşor mai riguros) mod de adresare registru-indirect (operand în memorie la adresa dată de registru). Exemple de instrucţiuni cu mod de adresare indirect:

XOR A, (R7) ;A←A ⊕ MEM$|_{R7}$. Conţinutul locaţiei de memorie adresată ;de R_7 reprezintă operandul sursă adresat indirect. Se face o ;funcţie *XOR* între conţinutul acumulatorului şi operandul sursă ;specificat şi rezultatul este depus în acumulator.

ST (R3), A ;MEM$|_{R3}$←A. Conţinutul acumulatorului este depus în locaţia ;de memorie adresată de R_3. Conţinutul locaţiei de memorie ;adresată de R_3 reprezintă operandul destinaţie adresat indirect.

(7),(9) • În cazul modului de adresare indexat (AX), adresa operandului se obţine adunând registrul general selectat de câmpul ADR. REG. (biţii $IR_{7\div4}$ din codul instrucţiunii) cu INDEX-ul codificat în codul instrucţiunii (câmpul $IR_{3\div0}$). Se calculează această adresă şi se încărcată în registrul ADR pentru a pregăti aducerea operandului din memorie (figura 2.5.d).). Exemple de instrucţiuni cu mod de adresare indexat:

AND A, 6(R1) ;A←A·MEM$|_{R1+6}$. Conţinutul locaţiei de memorie de la ;adresa R_1+6 reprezintă operandul sursă adresat indexat. Se ;face o funcţie *AND* între conţinutul acumulatorului şi ;operandul sursă specificat şi rezultatul este depus în ;acumulator.

ST 4(R5), A ;MEM$|_{R5+4}$←A. Conţinutul acumulatorului este depus în ;locaţia de memorie de la adresa R_5+4. Conţinutul locaţiei de ;memorie de la adresa R_5+4 reprezintă operandul destinaţie ;adresat indexat.

(10) • Se citeşte operandul din memorie şi se încarcă în registrul M.
(3),(6),(11) • Din faza OF se trece în faza de execuţie (EX) care începe cu impulsul de stare I_1.

2.6.2 Tabelul de comenzi aferent fazei OF

Tabelul de comenzi aferent fazei OF este redat în figura 2.20. Acest tabel de comenzi este o transpunere fidelă, în comenzi elementare, a operaţiilor din organigrama fazei OF.

Faza OF

	I_1	I_2
AM:	*PdIR[OP], SBUS, PdALU, PmM, TEX, TI_1*	*RD, PdMEM, PmM, TEX, TI_1*
AD:	*PdRG, DBUS, PdALU, PmM, TEX, TI_1*	
AI:	*PdRG, DBUS, PdALU, PmADR*	
AX:	*PdIR[IND], PdRG, SUM, PdALU, PmADR*	

Fig. 2.20 Tabelul de comenzi aferent fazei OF

Comanda *PdIR[OP]* emite pe SBUS octetul inferior din IR (operandul imediat codificat în codul instrucţiunii sub forma unei constante pe 8 biţi). Acest operand este un număr cu semn reprezentat în cod complementar (bitul IR_7 este bitul de semn iar câmpul $IR_{6\div0}$ reprezintă partea de mărime). În acest mod putem specifica atât costante pozitive cât şi constante negative. Cu alte cuvinte, putem specifica constante în domeniul $[-2^7, +2^7]$. Pentru că unitatea ALU revendică operanzi pe 16 biţi, operandul imediat este emis pe SBUS cu extensie de semn pe octetul superior, aşa cum indică figura 2.21. Având în vedere reprezentarea în cod complementar (complement faţă de 2), extensia de semn extinde operandul de la 8 la 16 biţi, dar conservă valoarea acestuia.

Comanda *PdIR[IND]* emite INDEX-ul pe SBUS. INDEX-ul este codificat pe 4 biţi în codul instrucţiunii (biţii $IR_{3\div0}$), fiind un număr cu semn reprezentat în cod complementar (bitul 3 este bit de semn). Emisia pe SBUS a indexului revendică o extensie de semn similară (figura 2.22).

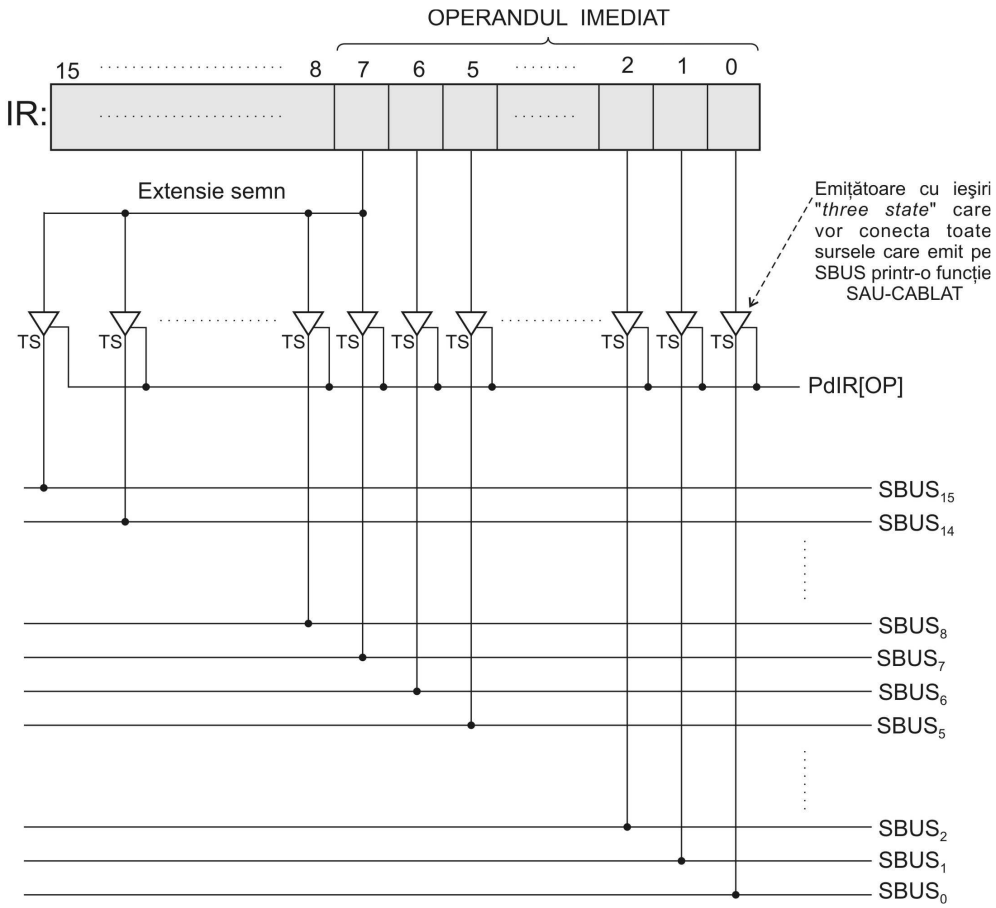


Fig. 2.21 Emisia operandului imediat din IR, cu extensie de semn

INDEX-ul, extins la o lungime de 16 biţi (cu conservarea valorii) va fi adunat în ALU cu un al doilea operand pe 16 biţi (registrul general specificat în instrucţiune). În acest mod va fi calculată adresa operandului, în cazul AX (blocul (9) din figura 2.19).

Observaţie:
În baza formatului de instrucţiune adoptat se observă că putem specifica constante într-un domeniu relativ restrâns, [-2^7, +2^7], şi indecşi într-un domeniu şi mai restrîns, [-2^3, +2^3]. Această restrângere de domeniu se datorează utilizării formatului de lungime fixă, pe 16 biţi. Dacă am fi recurs la un format pe 32 biţi, (lucru care se întâmplă la procesoarele pe 32 biţi) atunci constantele şi respectiv indecşii ar fi luat valori în domenii mult mai mari. Făcând însă abstracţie de această restrângere severă de domeniu, la nivel de principiu, modurile de adresare îşi păstrează semnificaţia nealterată. Ori, în această lucrare, urmărim explicarea conceptelor fundamentale de proiectare. Aceste concepte vor fi cu atât mai evidente pentru cititor, cu cât structura, pe care facem acest exerciţiu de proiectare, este mai simplă.

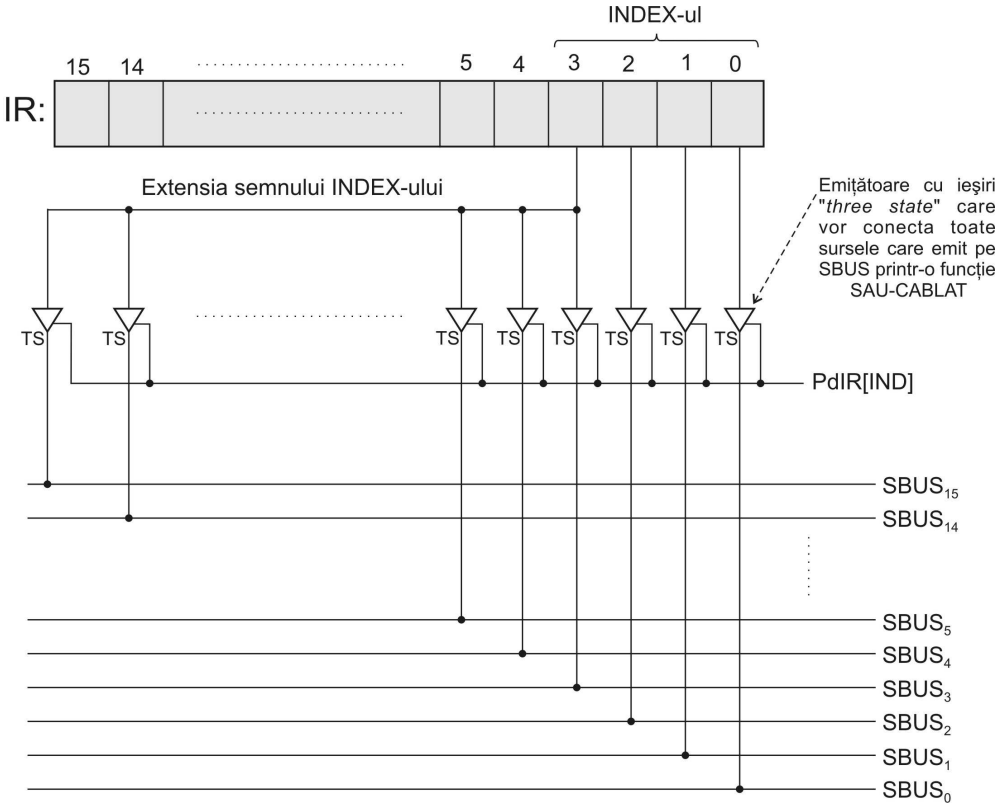


Fig. 2.22 Emisia INDEX-ului pe SBUS, cu extensie de semn

Pe baza tabelului de comenzi din figura 2.20 şi respectiv organigramei din figura 2.19, constatăm că faza OF se execută într-un singur impuls de stare (o singură perioadă de tact procesor) în cazurile AM şi AD şi respectiv, în două impulsuri în cazurile AI şi AX.

2.7 Proiectarea fazei EX

În faza EX se execută operaţia revendicată de instrucţiune, ca de exemplu:

- adunarea operanzilor, depunerea rezultatului în A şi setarea *flag*-urilor de condiţii în acord cu rezultatul, în cazul instrucţiunii ADD A, *src*.
- calculul adresei de salt şi încărcarea acesteia în PC, în cazul instrucţiunii JMP *adr*.
- salvarea registrului A în stivă, în cazul instrucţiunii PUSH A.
- validarea întreruperilor prin setarea *flag*-ului BVI, în cazul instrucţiunii EI.

Prin urmare, faza EX are un conţinut diferit de la o instrucţiune la alta (conţinut dependent de *OPCODE*-ul instrucţiunii). Fiecare instrucţiune are propria ei fază de execuţie.

Fazele IF şi respectiv OF au un conţinut standardizat, independent de tipul (*OPCODE*-ul) instrucţiunii. Faza IF nu depinde nici măcar de modul de adresare codificat în instrucţiune. Doar ieşirea din IF se poate face pe trei direcţii, în funcţie de clasa din care face parte instrucţiunea în discuţie:

- spre OF dacă instrucţunea este din clasa IOP.
- spre EX dacă înstrucţiunea este din clasa ID.
- spre INT dacă în IR s-a încărcat un cod ilegal.

Faza OF localizează operandul referit explicit (prin câmpurile MA şi respectiv ADR din codul instrucţiunii), îl citeşte şi îl încarcă în registrul tampon M. Prin urmare faza OF depinde doar de modul de adresare codificat în instrucţiune. Ea easte parcursă numai dacă instrucţiunea din IR este din clasa IOP. Decizia de a încărca operandul în M, oricare ar fi modul de adresare codificat în instrucţiune, am luat-o pentru a simplifica faza EX. Astfel, la intrarea în EX, vom şti că operandul se află în M. De aici trebuie preluat şi procesat aşa cum cere *OPCODE*-ul instrucţiunii. În acest mod, faza EX devine independentă de modul de adresare şi prin urmare, mai simplă.

2.7.1 Faza EX pentru instrucţiunile din clasa IOP

A. Instrucţiunile „*load*”

În clasa IOP identificăm trei instrucţiuni „*load*”: LD A,*src*; LD SP,*src*; LD FLAG,*src.* Instrucţiunile „*load*” au o execuţie foarte simplă (tabelul de comenzi din figura 2.23).

Faza EX – instrucţiuni „*load*”

I_1
PdM, DBUS, PdALU, PmX
$\overline{\text{INTR}}$: *TIF*, TI_1
INTR : *TINT*, TI_1

Fig. 2.23 Faza EX aferentă instrucţiunilor ***„load”*** – tabelul de comenzi.

În faza anterioară (OF), operandul *src* a fost localizat, citit şi încărcat în registrul M. În faza EX acesta este transferat din M în registrul destinaţie specificat în instrucţiune.

Fiecare dintre cele trei instrucţiuni are propriul tabel de execuţie, dar cele trei tabele sunt aproape identice. Este motivul pentru care în figura 2.23 am prezentat un singur tabel (generic) care, prin particularizarea comenzii generice *PmX*, va genera trei tabele.

Rezultă că:

- X=A, în cazul instrucţiunii LD A,*src* (*Pm X* devine, de fapt, *PmA*)
- X=SP, în cazul instrucţiunii LD SP,*src* (*Pm X* devine *PmSP*)
- X=FLAG, în cazul instrucţiunii LD FLAG,*src* (*Pm X* devine *PmFLAG*)

Să notăm că, la proiectarea unităţii de control (denumită şi bloc BGC), trebuie luate în considerare toate tabelele de comenzi. Prin urmare, pentru instrucţiunile „load" vor trebui luate în considerare trei tabele (fiecare instrucţiune cu propriul tabel) şi nu doar tabelul generic din figura 2.23.

La sfârşitul fazei EX se implementează punctul interuptibil care presupune testarea cererii globale de întrerupere INTR (vezi paragraful 2.3). Din punctul interuptibil:

- se revine în faza IF pentru *fetch*-ul următoarei instrucţiuni din program, dacă INTR=0 (nici-un periferic nu solicită intrerupere).
- se intră în faza de întrerupere INT, dacă INTR=1 (cel puţin un periferic solicită întrerupere).

În tabelul de comenzi, punctul interuptibil este implementat astfel:

$\overline{\text{INTR}}$: *TIF*, TI_1 ;dacă $\overline{\text{INTR}}$ =1 atunci „treci în faza IF" şi „treci în impulsul I_1"

INTR : *TINT*, TI_1 ; dacă INTR=1 atunci „treci în faza INT" şi „treci în impulsul I_1"

Comenzile *TIF* şi respectiv *TINT* sunt comenzi pentru generatorul de faze (GF). Acestea vor determina comutarea acestuia în faza precizată de comandă. După cum indică cronograma din figura 2.17, această comutare se va realiza la finele impulsului de stare curent (perioadei de tact curente). Comanda TI_1 este o comandă pentru generatorul impulsurilor de stare (GIS). Aceasta va iniţializa GIS-ul (prin comanda TI_1, GIS-ul trece din impulsul de stare curent în impulsul I_1, oricare ar fi impulsul curent).

În tabelul de comenzi, o variabilă (exprsie) urmată de caracterul „:" semnifică faptul că acea variabilă (expresie) reprezintă o condiţie testată. Comenzile care urmează după caracterul „:" vor fi activate numai dacă condiţia respectivă are valoarea 1 (dacă este îndeplinită). Dacă condiţia are valoarea 0, comenzile în discuţie nu se activează.

B. Instrucţiunea ADD A,*src*

Instrucţiunea ADD A,*src* trebuie să realizeze operaţia A←A+*src* (cu setarea *flag*-urilor de condiţii în acord cu rezultatul depus în A). În faza anterioară (faza OF) operandul *src* a fost încărcat în registrul M. Prin urmare, în faza EX trebuie executată de fapt operaţia A←A+M (cu setarea *flag*-urilor de condiţii). Organigrama fazei EX aferentă acestei instrucţiuni este redată în figura 2.24. Din organigramă se deduce că instrucţiunea ADD A,*src* se execută într-un singur impuls de stare (într-o singură perioadă de tact). Tabelul de comenzi aferent acestei execuţii este redat în figura 2.25.

Să observăm că punctul interuptibil a fost rescris într-o manieră optimizată; comanda TI_1 este în fond necondiţionată (nu depinde de condiţia INTR). Din acest motiv am repoziţionat-o în tabel, în lista comenzilor necondiţionate; în baza convenţiei adoptate, pe o coloaoană dintr-un tabel de comenzi, putem avea două categorii de comenzi: condiţionate şi respectiv, necondiţionate. Completarea fiecărei coloane începe cu lista comenzilor necondiţionate.

Setarea *flag*-urilor de condiţii (C=*carry*, Z=*zero*, S=*sign*, V=*overflow*) se face pe baza semnalelor de condiţii generate de ALU. ALU este un bloc pur combinaţional. În consecinţă, semnalele de condiţii sunt stabile şi corecte atâta timp cât operanzii sunt stabili pe intrările ALU şi respectiv comanda pentru operaţia ALU (în cazul de faţă comanda *SUM*) este activă. Cum aceste condiţii sunt îndeplinite doar pe durata impulsului de stare I_1 din cadrul fazei EX (pe durata unei singure perioade de tact), rezultă că în cadrul aceluiaşi impuls trebuie poziţionate şi *flag*-urile de condiţii.

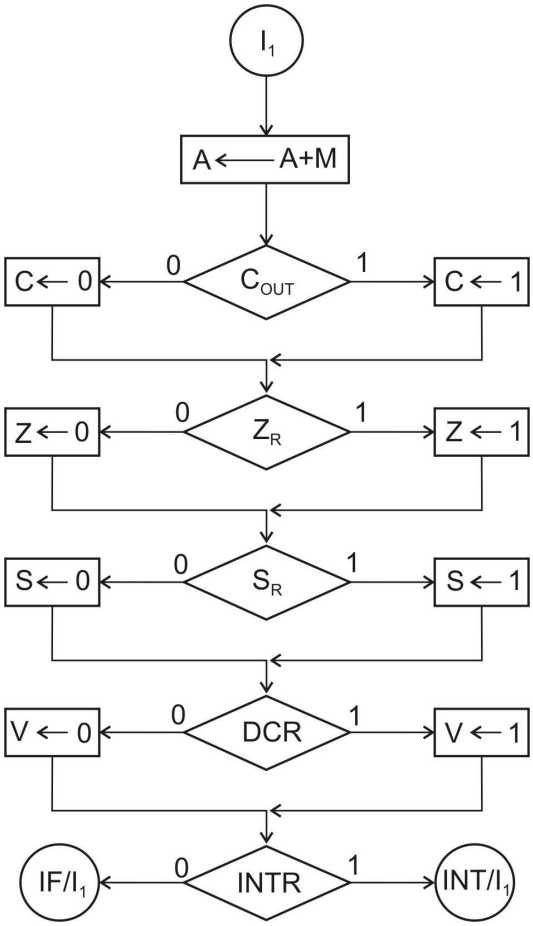


Fig 2.24 Organigrama fazei EX aferentă instrucţiunii **ADD A,*src***

Faza EX – instrucţiunea ADD A,*src*

I_1
PdA, PdM, SUM, PdALU, PmA, TI_1
C_{out} : *A(1)C*
$\overline{C}_{out}$: *A(0)C*
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
DCR_{ADD} : *A(1)V*
$\overline{DCR}_{ADD}$: *A(0)V*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.25 Faza EX aferentă instrucţiunii **ADD A,*src*** – tabelul de comenzi

Cele patru semnale de condiţii testate pentru setarea *fag*-urilor sunt:

- C_{out} –transport final (transport de ieşire din rangul cel mai semnificativ) care apare la sumatorul pe 16 biţi ce intră în componenţa ALU; utilizat pentru setarea bistabilului C din registrul FLAG.

- Z_R –rezultat egal cu zero pe ieşirile ALU; utilizat pentru setarea bistabilului Z. Semnalul Z_R se generează simplu din cei 16 biţi ai rezultatului obţinut pe ieşirile ALU. Dacă notăm cu $ALU_{15\div0}$ cei 16 biţii ai rezultatului obţinut pe ieşirea blocului ALU, atunci ecuaţia condiţiei Z_R va fi:

$$Z_R = \overline{ALU_{15} + ALU_{14} + \cdots + ALU_1 + ALU_0} \qquad (2.2)$$

  Cu alte cuvinte, Z_R=1 numai dacă toţi biţii din componenţa rezultatului sunt zero (inclusiv ALU_{15} care reprezintă bitul de semn). Implementarea *hardware* a acestei condiţii este prezentată în figura 2.26.

- S_R –semnul rezultatului; utilizat pentru setarea *flag*-ului S. S_R este de fapt bitul ALU_{15} (bitul cel mai semnificativ din componenţa rezultatului). Implementarea *hardware* a acestei condiţii este prezentată în figura 2.26.

- DCR –depăşirea capacităţii de reprezentare la rezultat (rezultatul nu mai poate fi reprezentat pe 16 biţi deoarece iese din domeniul de reprezentare, revendicând un al 17-lea bit pentru reprezentarea sa corectă). În consecinţă, rezultatul va fi încărcat în registrul A trunchiat şi va fi setat *flag*-ul V. V=1

înseamnă rezultat incorect. Ecuaţia (2.3) sintetizează condiţia DCR pentru operaţia de adunare în cod complementar:

$$DCR_{ADD}=\underbrace{(S_A \odot S_M)}_{A}\bullet\underbrace{(S_R \oplus C_{out})}_{B} \qquad (2.3)$$

unde: S_A –semn acumulator (bitul A_{15}); semnul operandului 1.
S_M –semn registru M (bitul M_{15}); semnul operandului 2.
$\oplus$ –funcţia XOR (SAU EXCLUSIV)
$\odot$ –funcţia COINCIDENŢĂ (XOR NEGAT)

Paranteza B sintetizează condiţia <u>*DCR propriu-zis*</u> pentru operaţia de adunare în cod complementar. Nu prezentăm aici demonstraţia matematică aferentă. Această condiţie *DCR propriu-zis* este „amendată” cu paranteza A care înseamnă <u>operanzi de acelaşi semn</u>. Într-adevăr, la adunare nu se poate obţine depăşire decât dacă operanzii sunt de acelaşi semn (ambii pozitivi, caz în care se poate ieşi din domeniul de reprezentare spre dreapta, sau ambii negativi, caz în care se poate ieşi din domeniul de reprezentare spre stânga).

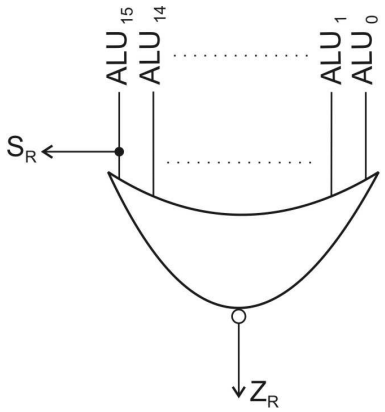


Fig 2.26 Implementarea semnalelor de condiţii Z_R şi S_R în *hardware*

Pentru setarea/resetarea *flag*-urilor sunt utilizate comenzi individuale de setare/resetare:

A(1)C –„adu la 1 *Carry*”	*A(0)C* –„adu la 0 *Carry*”
A(1)Z –„adu la 1 *Zero*”	*A(0)Z* –„adu la 0 *Zero*”
A(1)S –„adu la 1 *Sign*”	*A(0)S* –„adu la 0 *Sign*”
A(1)V –„adu la 1 *Overflow*”	*A(0)V* –„adu la 0 *Overflow*”

<u>Registrul FLAG</u>:
Registrul FLAG conţine bistabilii de condiţii (C/Z/S/V) şi bistabilul de validare a întreruperilor (BVI). În terminologia de specialitate bistabilul BVI este de regulă denumit

IE (*Interrupt Enable*) sau IF (*Interrupt enable Flag*). Structura pe care o propunem pentru registrul FLAG este redată în figura 2.27.

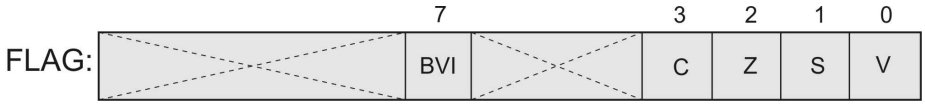
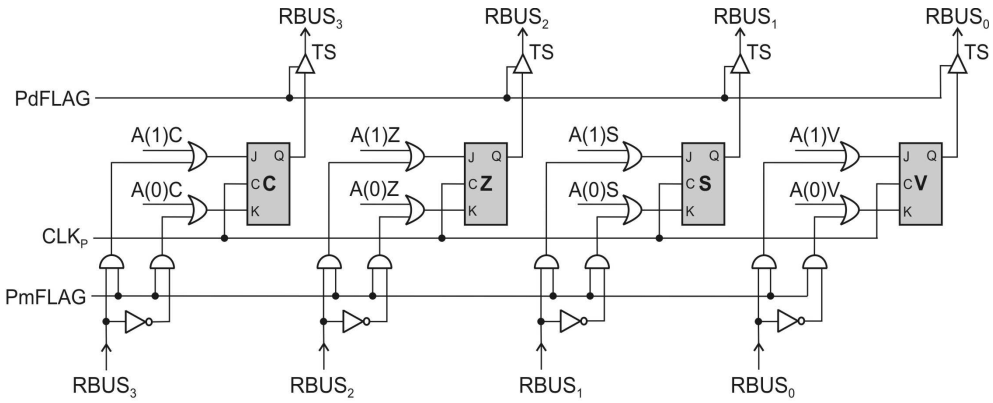


Fig 2.27 Structura registrului FLAG

După cum rezultă din schema bloc a procesorului didactic (figura 2.2), registrul FLAG poate fi încărcat cu date preluate de pe RBUS, prin activarea comenzii *PmFLAG*, şi respectiv poate fi emis pe RBUS, prin activarea comenzii *PdFLAG*. De asemenea, cei patru bistabili de condiţii sunt setaţi/resetaţi prin comenzi individuale de setare/resetare: (*A(1)C*, *A(1)Z*, *A(1)S*, *A(1)V*)/(*A(0)C*, *A(0)Z*, *A(0)S*, *A(0)V*). O implementare posibilă a celor patru bistabili de condiţii este prezentată în figura 2.28.



$RBUS_{3\div0}$-cele mai putin semnificative 4 linii ale RBUS-ului

Fig 2.28 Implementarea *flag*-urilor C/Z/S/V cu bistabili *J-K master-slave*

Relaţiile (2.4) redau ecuaţiile intrărilor J şi respectiv K aferente celor patru bistabili de condiţii:

$$\begin{cases} J_C = A(1)C + RBUS_3 \bullet PmFLAG & K_C = A(0)C + \overline{RBUS_3} \bullet PmFLAG \\ J_Z = A(1)Z + RBUS_2 \bullet PmFLAG & K_Z = A(0)Z + \overline{RBUS_2} \bullet PmFLAG \\ J_S = A(1)S + RBUS_1 \bullet PmFLAG & K_S = A(0)S + \overline{RBUS_1} \bullet PmFLAG \\ J_V = A(1)V + RBUS_0 \bullet PmFLAG & K_V = A(0)V + \overline{RBUS_0} \bullet PmFLAG \end{cases} \quad (2.4)$$

$RBUS_i$ reprezintă starea (*low/high*) a liniei de bus respective; cu alte cuvinte, linia de bus poate fi, în momentul respectiv, pe zero (*low*) sau pe unu (*high*). „Momentul respectiv" înseamnă desigur momentul în care apare tranziţia *low-high* pe intrarea C a bistabilului (frontul urcător al semnalului de tact aferent procesorului CLK_P), deoarece acest front urcător eşantionează intrările J şi K şi le memorează în bistabilul master.

C. Instrucţiunea SUB A,*src*

Calculatorul operează cu numere reprezentate în cod complementar (complement faţă de 2). În aceste condiţii operaţia de scădere se transformă în adunare: se adună la descăzut complementul faţă de doi al scăzătorului. Rezultatul va fi corect atâta timp cât nu iese din domeniu de reprezentare. Ieşirea din domeniu va trebui semnalată prin setarea *flag*-ului *overflow*. Nu facem aici demonstraţia matematică a acestei reguli de efectuare a operaţiei de scădere în cod complementar; ea poate fi găsită în literatura de specialitate. Prin urmare, adunarea şi scăderea se vor executa cu ajutorul unui **sumator** pe 16 biţi (un singur bloc în componenţa ALU pentru ambele instrucţiuni). Se obţine astfel o simplificare structurală a unităţii ALU. Dacă sistemul de calcul ar fi operat în cod direct atunci:

- Scăderea ar fi revendicat un dispozitiv propriu în structura ALU, numit scăzător.

- Semnul rezultatului s-ar fi stabilit pe baza unor reguli mult mai complexe şi care ar fi revendicat o implementare în *hardware* evident mai complexă, inclusiv pentru operaţia de adunare. De exemplu, dacă se adună două numere de semne diferite, reprezentate în cod direct, în realitate trebuie scăzute modulele acestora iar semnul rezultatului va fi semnul operandului al cărui modul are valoarea mai mare. Prin urmare, pentru stabilirea semnului rezultatului trebuie comparate modulele operanzilor.

Codul complementar elimină aceste complicaţii şi conduce la simplificarea unităţii ALU.

Ca şi instrucţiunea ADD A,*src*, instrucţiunea SUB A,*src* se va executa într-un singur impuls de stare (într-un singur tact). Tabelul de comenzi aferent acestei execuţii este redat în figura 2.29.

Conform aşteptărilor, constatăm un grad mare de similaritate între tabelele fazelor de execuţie aferente celor două instrucţiuni (SUB A,*src* şi ADD A,*src*). La nivelul comenzilor elementare, diferenţele sunt:

- $Pd\overline{M}$ -emite pe DBUS conţinutul registrului M inversat bit cu bit. Prin urmare, la instrucţiunea SUB, pe DBUS va fi emis complementul faţă de 1 al scăzătorului.

- C_{in} -apare suplimentar la SUB, aplicând un 1 pe intrarea de transport a unităţii ALU.

($Pd\overline{M}$ şi C_{in}), în conjuncţie cu comanda *SUM*, vor realiza complementul faţă de 2 al scăzătorului preluat din registrul M. Acest complement va fi adunat la descăzut (la conţinutul registrului A).

Faza EX – instrucţiunea SUB A,*src*

I_1
$PdA,\ Pd\overline{M}\ ,\ C_{in},\ SUM,\ PdALU,\ PmA,\ TI_1$
C_{out} : $A(1)C$
$\overline{C}_{out}$: $A(0)C$
Z_R : $A(1)Z$
$\overline{Z}_R$: $A(0)Z$
S_R : $A(1)S$
$\overline{S}_R$: $A(0)S$
DCR_{SUB} : $A(1)V$
$\overline{DCR}_{SUB}$: $A(0)V$
$\overline{INTR}$: TIF
$INTR$: $TINT$

Fig 2.29 Faza EX aferentă instrucţiunii **SUB A,*src*** – tabelul de comenzi

Reamintim regula practică de calcul, în două etape succesive, a complementului faţă de 2 (în baza căreia se calculează complementul scăzătorului preluat din M):

1. se complementează reprezentarea în cod direct a numărului în cauză, bit cu bit.
2. se adună o unitate pe rangul cel mai puţin semnificativ.

Comenzile $Pd\overline{M}$ şi C_{in}, în conjuncţie cu comanda *SUM*, realizează, într-o singură etapă, această complementare.

O ultimă deosebire se referă la expresia condiţiei de depăşire de domeniu la scădere:

$$DCR_{SUB}=(S_A \oplus S_M)\bullet(S_R \oplus C_{out}) \qquad (2.5)$$

Spre deosebire de adunare (ecuaţia 2.3), la scădere poate rezulta depăşire de domeniu la rezultat (ecuaţia 2.5), numai dacă operanzii sunt de semne diferite ($S_A \oplus S_M=1$).

D. Instrucţiunea CMP A,*src*

Instrucţiunea CMP A,*src* este aproape identică cu instrucţiunea SUB A,*src* (comparaţia se face prin scădere). Prin urmare, se preia tabelul de comenzi de la instrucţiunea SUB A,*src,* cu o singură modificare: se elimină comanda *PmA*. Demonstraţia este foarte simplă: compararea se face prin scădere, în scopul poziţionării *flag*-urilor de condiţii; rezultatul nu trebuie depus la destinaţie. Tabelul de comenzi aferent acestei execuţii este redat în figura 2.30.

Faza EX – instrucţiunea CMP A,*src*

I_1
PdA, $Pd\overline{M}$, C_{in}, *SUM*, *PdALU*, TI_1
C_{out} : *A(1)C*
$\overline{C}_{out}$: *A(0)C*
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
DCR_{SUB} : *A(1)V*
$\overline{DCR}_{SUB}$: *A(0)V*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.30 Faza EX aferentă instrucţiunii **CMP A,*src*** – tabelul de comenzi

Observaţie:
Comanda PdALU emite pe RBUS rezultatul obţinut pe ieşirea ALU. Această comandă nu are nici-o acţiune relativă la poziţionarea flag-urilor de condiţii şi prin urmare ar trebui eliminată din tabelul de comenzi aferent instrucţiunii CMP A,src. Să remarcăm totuşi că execuţia instrucţiunii este corectă chiar dacă se menţine această comandă în tabel; rezultatul comparaţiei (scăderii) este emis pe RBUS dar, în lipsa comenzii PmA, nu este depus la destinaţie. Fireşte că ar fi preferabilă varianta fără PdALU deoarece conduce la o mică simplificare în hardware.

E. Instrucţiunile logice

În clasa IOP identificăm trei instrucţiuni logice: AND A,*src*; XOR A,*src*; OR A,*src*. Sunt practic similare cu instrucţiunile aritmetice cu deosebirea că ALU va face, nu o operaţie aritmetică, ci una logică. De asemenea, la instrucţiunile logice, *flag*-urile C şi V sunt lipsite practic de semnificaţie (nu poate rezulta C_{out}=1 sau *DCR*=1 la o operaţie logică, oricare ar fi aceasta). Prin urmare, la instrucţiunile logice, nu are sens poziţionarea *flag*-urilor C şi respectiv V.

Flag-ul S poate rezulta în 1, în urma unei operaţii logice, chiar dacă îşi pierde semnificaţia generică de semn al rezultatului (instrucţiunile logice operează cu valori logice şi nu cu operanzi numerici care conţin bit de semn şi parte de mărime). În *flag*-ul S se va memora, ca şi la instrucţiunile aritmetice, bitul cel mai semnificativ din componenţa rezultatului (bitul ALU_{15}).

Flag-ul Z îşi păstrează semnificaţia nealterată şi trebuie setat dacă toţi biţii rezultatului sunt zero; în caz contrar trebuie resetat.

În concluzie, la instrucţiunile logice se impune poziţionarea doar a două *flag*-uri: S şi respectiv Z. Tabelul de comenzi aferent fazei de execuţie a instrucţiunii AND A,*src* este redat în figura 2.31.

Faza EX – instrucţiunea AND A,*src*

I_1
PdA, PdM, AND, PdALU, PmA, TI_1
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.31 Faza EX aferentă instrucţiunii **AND A,*src*** – tabelul de comenzi

Tabelele de comenzi aferente instrucţiunilor XOR A,*src* şi OR A,*src* sunt similare celui aferent instrucţiunii AND A,*src*, cu următoarerle diferenţe:

- la XOR A,*src* comanda *AND* se înlocuieşte cu comanda *XOR*.
- la OR A,*src* comanda *AND* se înlocuieşte cu comanda *OR*.

La proiectarea blocului de generare a comenzilor (BGC) trebuie luate în considerare şi tabelele de comenzi aferente instrucţiunilor XOR şi OR ! Comenzile *AND*, *XOR* şi *OR* sunt desigur comenzi pentru ALU; la activarea unei asemenea comenzi, ALU va executa operaţia logică specificată. În structura ALU vor exista trei blocuri logice simple (AND, XOR şi OR), activate selectiv, de comenzile *AND*, *XOR* şi *OR*. Blocul AND va fi format din 16 porţi ŞI, blocul XOR din 16 porţi SAU-EXCLUSIV şi blocul OR din 16 porţi SAU. Toate porţile vor fi cu câte două intrări.

F. Instrucţiunile INC *dest* şi DEC *dest*

Instrucţiunile INC *dest* şi respectiv DEC *dest* sunt instrucţiuni de incrementare/decrementare cu 1 (instrucţiuni aritmetice cu un singur operand). Incrementarea/decrementarea operandului înseamnă în fond modificarea valorii acestuia iar modificarea valorii presupune o operaţie de scriere şi din acest motiv operandul nu va mai avea semnificaţia de operand sursă ci de operand destinaţie. Suplimentar, INC *dest* şi DEC *dest* sunt instrucţiuni aritmetice şi prin urmare vor seta şi *flag*-urile de condiţii în acord cu rezultatul incrementării/decrementării.

Faptul că operandul are semnificaţia de operand destinaţie şi nu de operand sursă va induce anumite schimbări în raport cu fazele de execuţie descrise până în acest punct. Operandul sursă trebuie doar citit. Operandul destinaţie trebuie citit, procesat şi în final rescris. Citirea operandului se face în faza OF; procesarea şi rescrierea noii valori trebuie realizată în faza EX.

Tabelele de comenzi aferente fazei de execuţie a instrucţiunilor INC *dest* şi DEC *dest* sunt redate în figurile 2.32 şi respectiv 2.33.

Faza EX – instrucţiunea INC *dest*

I_1
$Pd0_S$, PdM, C_{in}, SUM, PdALU, TI_1
AD : *PmRG*
AI : *WR*
AX : *WR*
C_{out} : *A(1)C*
$\overline{C}_{out}$: *A(0)C*
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
DCR_{ADD} : *A(1)V*
$\overline{DCR}_{ADD}$: *A(0)V*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.32 Faza EX aferentă instrucţiunii **INC *dest*** – tabelul de comenzi

Faza EX – instrucţiunea DEC *dest*

I_1
$Pd\text{-}1_S$, PdM, SUM, PdALU, TI_1
AD : *PmRG*
AI : *WR*
AX : *WR*
C_{out} : *A(1)C*
$\overline{C}_{out}$: *A(0)C*
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
DCR_{SUB} : *A(1)V*
$\overline{DCR}_{SUB}$: *A(0)V*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.33 Faza EX aferentă instrucţiunii **DEC *dest*** – tabelul de comenzi

Analizând aceste tabele constatăm că, pentru execuţia instrucţiunilor INC *dest* şi respectiv DEC *dest*, sunt necesare următoarele operaţii (concurente deoarece, aşa cum indică tabelele de comenzi, execuţia se face pe un singur impuls de stare):

- incrementarea (decrementarea) cu 1 a registrului M. Reamintim faptul că operandul *dest* a fost citit şi încărcat în M în faza OF (faza anterioară). Incrementarea (decrementarea) se va face în ALU şi cu această ocazie vor fi setate şi *flag*-urile de condiţii.

- transferul valorii obţinute prin incrementare (decrementare) în operandul *dest*. Acest transfer revendică o operaţie de scriere într-un registru general sau într-o locaţie de memorie, în funcţie de modul de adresare utilizat în instrucţiune:

  - În cazul AD (adresare directa) instrucţiunile ar fi de forma INC R_i / DEC R_i, unde R_i poate fi orice registru general. Rezultatul incrementării/ decrementării trebuie depus deci în registrul general specificat în instrucţiune. Registrul specificat este automat selectat, în cadrul setului de regiştrii generali, cu ajutorul câmpului ADR. REG. din codul instrucţiunii, câmp ce se aplică pe intrările de adresă aferente matricii de memorare ce găzduieşte setul de regiştri generali (vezi explicaţiile de la 2.6.A). În consecinţă, pentru depunerea rezultatului mai trebuie activată doar comanda generală de scriere în setul de regiştrii: *PmRG*.

  - În cazul AI (adresare indirectă) instrucţiunile ar fi de forma INC (R_i) / DEC (R_i). Rezultatul incrementării/decrementării trebuie depus deci în memorie la adresa furnizată de registrul R_i. În faza OF a instrucţiunii, adresa operandului a fost transferată din R_i în registrul ADR în vederea extragerii operandului din memorie (blocul (8) din figura 2.19). În faza EX, această adresă se află încă în ADR. Prin urmare, pentru depunerea rezultatului, în cadrul fazei EX pe care o discutăm, mai trebuie activată doar comanda de scriere în memorie (*WR*). Cu *PdMEM*=0 (inactivă), datele de scris vor fi transferate de pe RBUS pe MBUS (vezi figura 2.36), de unde vor fi scrise în locaţia selectată de adresa existentă în ADR.

  - Cazul AX (adresare indexată) este similar cu cazul AI. Rezultatul trebuie depus într-o locaţie de memorie, a cărei adresă a fost calculată în faza OF şi încărcată în ADR (blocul (9) din figura 2.19). Ca în cazul AI, pentru depunerea rezultatului, în cadrul fazei EX mai trebuie activată doar comanda de scriere în memorie (*WR*).

    Forma instrucţiunii în cazul AX ar fi INC 7(R_i) / DEC 4(R_i). Constantele exemplificate (7 şi respectiv 4) sunt indecşi pentru codificarea cărora am rezervat 4 biţi în formatul instrucţiunilor din clasa IOP (figura 2.5.d).). Modul de adresare indirect (AI) ar putea fi privit ca un caz particular al modului de adresare indexat (AX); atunci când indexul are valoarea zero, modul AX devine AI.

- În cazul AM instrucţiunile de incrementare/decrementare ar fi de forma: INC *ct* / DEC *ct*, unde *ct* reprezintă o constantă. Dată fiind inutilitatea unor asemenea instrucţiuni, vom considera că INC *dest* / DEC *dest* cu adresare imediată reprezintă cod ilegal (CIL). Condiţia CIL a fost detectată global şi tratată în faza IF.

Să observăm că poziţionarea *flag*-urilor la instrucţiunea INC *dest* se face ca la instrucţiunea de adunare iar la instrucţiunea DEC *dest* se face ca la instrucţiunea de scădere.

Comanda $Pd0_S$ emite constanta zero pe SBUS. Implementarea acestei comenzi în *hardware* este redată în figura 2.34.

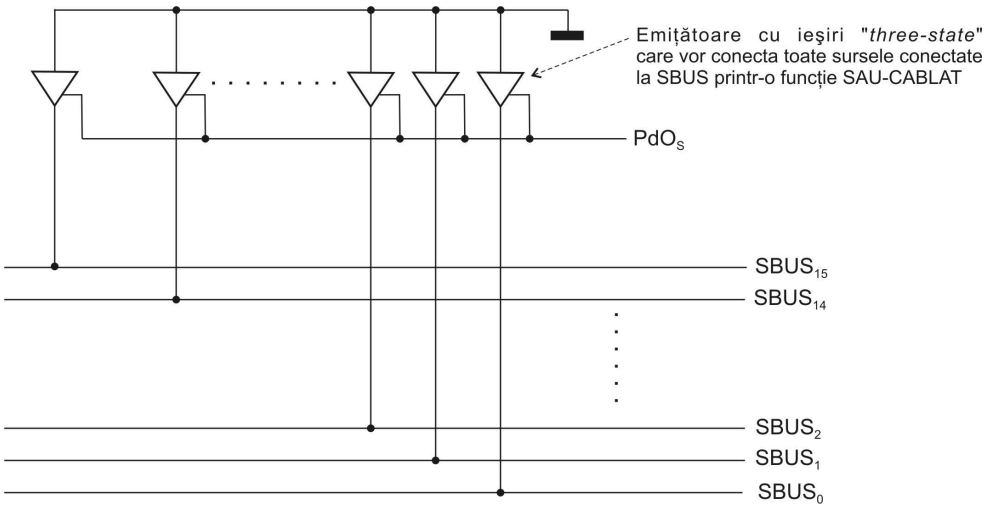


Fig 2.34 Implementarea comenzii $Pd0_S$ în *hardware*

Comanda $Pd\text{-}1_S$ emite pe SBUS constanta -1, reprezentată pe 16 biţi în cod complementar:

$$(-1)_C=1111111111111111_2 \qquad (2.6)$$

Implementarea acestei comenzi în *hardware* este redată în figura 2.35.

În cazurile AIş şi AX, operandul *dest* incrementat (decrementat) este rescris în memorie cu ajutorul comenzii *WR* (fig 2.32 şi 2.33). Incrementarea (decrementarea) se face în ALU şi cu ajutorul comenzii *PdALU*, rezultatul (incrementării/decrementării) este emis pe RBUS. Cu *PdMEM*=0 (inactivă), acest rezultat va fi transferat de pe RBUS pe MBUS, de unde va fi scris în locaţia de memorie selectată de adresa existentă în ADR. În consecinţă, conexiunea dintre cele două busuri, MBUS şi respectiv RBUS, trebuie să fie bidirecţională, sensul fiind controlat de comanda *PdMEM* (figura 2.36).

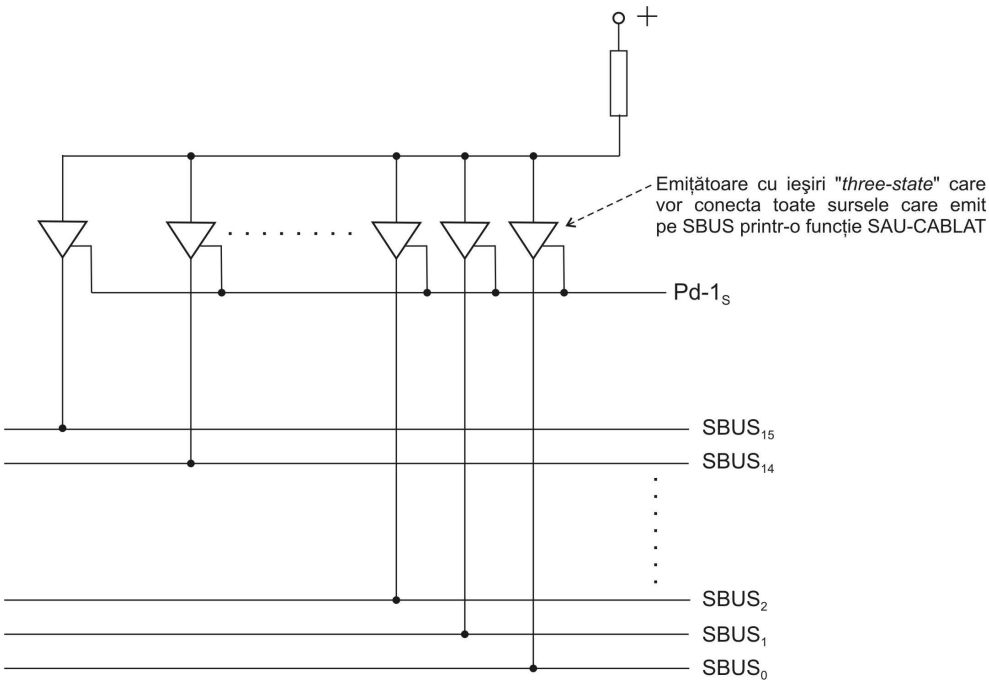


Fig 2.35 Implementarea comenzii *Pd-1$_S$* în *hardware*

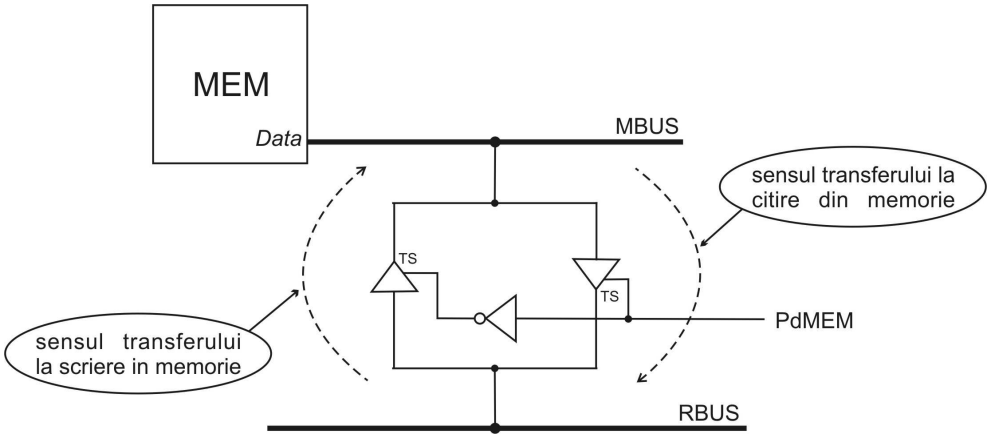


Fig 2.36 Conexiunea bidirecţională MBUS – RBUS

PdMEM=1 realizează conexiunea MBUS→RBUS, necesară la operaţia de citire din memorie. Este motivul pentru care, o comandă *RD* (citire din

memorie) este întotdeauna acompaniată, în tabelele de comenzi, de o comandă *PdMEM*.

PdMEM=0 realizează conexiunea RBUS→MBUS, necesară la operaţia de scriere în memorie. Este motivul pentru care o comandă *WR* (scriere în memorie) nu este niciodată însoţită de o comandă *PdMEM*, în tabelele de comenzi.

G. Instrucţiunile PUSH *reg* şi POP *reg*

Instrucţiunea PUSH *reg* salvează (memorează) conţinutul registrului specificat în stivă. Instrucţiunea POP *reg* are o acţiune inversă; restaurează registrul specificat din stivă (citeşte conţinutul locaţiei din vârful stivei şi încarcă acest conţinut în registru). Stiva ocupă practic o zonă de memorie iar pentru adresarea locaţiilor stivei este utilizat registrul SP (*Stack Pointer*). Iniţializarea stivei înseamnă, în fapt, încărcarea unei adrese în SP. Această adresă va reprezenta vârful iniţial al stivei care, imediat după iniţializare, coincide cu baza stivei. Vârful stivei se modifică la orice *push*/*pop* în/din stivă; baza rămâne cea stabilită la iniţializare.

Instrucţiunile PUSH *reg* şi POP *reg* se execută principial în doi paşi:

PUSH *reg*:	1. SP←SP-2	;decrementează *pointer*-ul de stivă cu 2 (adaugă o ;nouă locaţie liberă în vârful stivei).
	2. MEM $\vert_{SP}$←*reg*	;memorează conţinutul registrului specificat în ;noua locaţie adăugată în vârful stivei.
POP *reg*:	1. *reg*← MEM $\vert_{SP}$	;citeşte conţinutul locaţiei din vârful stivei şi ;încarcă datele citite în registrul specificat.
	2. SP←SP+2	;incrementează *pointer*-ul de stivă cu 2 (după ;citirea locaţiei, aceasta se elimină din stivă, ;fixându-se noul vârf al stivei).

În concluzie, stiva evoluează (prin depunere) în jos (spre adrese descrescătoare), funcţionează după principiul LIFO (*Last-In First-Out*) iar registrul SP conţine în permanenţă adresa vârfului stivei (adresa ultimului element introdus în stivă cu *push* şi încă neextras cu *pop*).

Tabelul de comenzi aferent fazei de execuţie a instrucţiunii PUSH *reg* este redat în figura 2.37 iar cel aferent instrucţiunii POP *reg* este redat în figura 2.38.

Instrucţiunea PUSH *reg* se execută în trei impulsuri de stare (perioade de tact) succesive:

I_1: SP←SP-2 (decrementare cu 2 a registrului SP).

I_2: ADR←SP (adresa noului vârf al stivei).

I_3: MEM $\vert_{ADR}$←*reg* (conţinutul registrului este scris în stivă)

Instrucţiunea POP *reg* se execută în două impulsuri de stare (perioade de tact) succesive:

I_1: ADR←SP (adresa vârfului stivei).

I_2: *reg*←MEM$|_{ADR}$ (citeşte locaţia din vârful stivei şi încarcă datele citite în registru).
SP←SP+2 (incrementare cu 2 a registrului SP pentru a fixa noul vârf al stivei).

Faza EX – instrucţiunea PUSH *reg*

I_1	I_2	I_3
-2SP	*PdSP, SBUS, PdALU, PmADR*	*PdRG, DBUS, PdALU, WR, TI_1* $\overline{INTR}$: *TIF* INTR : *TINT*

Fig 2.37 Faza EX aferentă instrucţiunii **PUSH *reg*** – tabelul de comenzi

Faza EX – instrucţiunea POP *reg*

I_1	I_2
PdSP, SBUS, PdALU, PmADR	*RD, PdMEM, PmRG, +2SP, TI_1* $\overline{INTR}$: *TIF* INTR : *TINT*

Fig 2.38 Faza EX aferentă instrucţiunii **POP *reg*** – tabelul de comenzi

Observaţii:

1. *Forma uzuală a instrucţiunilor push şi pop este PUSH R_i şi respectiv POP R_i. Prin urmare, operandul instrucţiunii este întotdeauna un registru. Asta înseamnă că instrucţiunile push şi pop au semnificaţie utilă numai cu mod de adresare direct (AD). Push sau pop cu adresare imediată, indirectă sau indexată n-au utilitate. Noi vom considera aceste cazuri ca fiind coduri ilegale (CIL), cazuri detectate global şi tratate în faza IF.*

2. *În cazul instrucţiunii PUSH reg, comanda PdRG, activată pe impulsul I_3, emite pe DBUS datele de scris în stivă. În faza OF, anterior parcursă, aceleaşi date au fost încărcate în registrul M (blocul (5) din organigrama 2.19). Rezultă că parcurgerea fazei OF, în cazul instrucţiunii PUSH R_i, este practic inutilă. Inutilitatea fazei OF este şi mai evidentă în cazul instrucţiunii POP R_i deoarece, această instrucţiune revendică o operaţie de scriere în*

registru R_i (nu de citire); sunt scrise în R_i datele citite din memorie (din stivă). Să observăm că parcurgerea fazei OF nu provoacă erori în procesarea celor două instrucţiuni dar este inutilă (induce penalităţi de perfofmanţă). Proiectarea poate fi rafinată în sensul eliminării fazei OF din ciclul instrucţiunilor PUSH R_i şi respectiv POP R_i, aşa cum se procedează la instrucţiunile din clasa ID.

H. Instrucţiunile „store”

În clasa IOP identificăm trei instrucţiuni „*store*”: ST *dest*,A; ST *dest*,SP; ST *dest*,FLAG. Aceste instrucţiuni memorează registrul specificat (A/SP/FLAG) în operandul *dest* adresat explicit prin câmpurile MA şi respectiv ADR din codul instrucţiunii. Vom analiza variantele posibile pe care le putem identifica la nivelul instrucţiunilor „*store*”, luând ca exemplu instrucţiunea ST *dest*,A. În funcţie de modul de adresare utilizat, sunt posibile următoarele forme pentru instrucţiunea ST *dest*,A:

- În cazul AD (adresare directa) instrucţiunea ST *dest*,A ar fi de forma ST R_i,A, unde R_i poate fi orice registru general. Pentru acest caz, la nivelul asamblorului, mai sugestivă ar fi mnemonica MOV R_i,A , deoarece avem de-a face, de fapt, cu o instrucţiune „*move* registru-registru” (instrucţiune utilă, trebuie implementată, nu operează însă în memorie).

- În cazul AI (adresare indirectă) instrucţiunea ST *dest*,A ar fi de forma ST (R_i),A. Este un „*store*” plin de semnificaţie, deoarece registrul A va fi memorat în locaţia de memorie selectată de adresa furnizată de registrul R_i.

- În cazul AX (adresare indexată) instrucţiunea ST *dest*,A ar fi de forma ST 6(R_i),A , unde prin constanta 6 am exemplificat un index care este reprezentat în cod binar în codul instrucţiunii (vezi schema de principiu prezentată în figura 2.5.c).). Registrul A va fi depus tot într-o locaţie de memorie. Adresa locaţiei care va prelua conţinutul registrului A este: R_i+6. Din nou avem de-a face cu un „*store*” plin de semnificaţie.

- În cazul AM instrucţiunea ST *dest*,A ar fi de forma ST *ct*,A , unde *ct* reprezintă o constantă. Memorarea unui registru într-o constantă reprezintă un non-sens şi prin urmare acest caz va fi considerat cod ilegal (CIL). Condiţia CIL a fost detectată global şi tratată în faza IF.

Tabelele de comenzi aferente fazei de execuţie a instrucţiunilor ST *dest*,A / ST *dest*,SP / ST *dest*,FLAG sunt prezentate în figurile 2.39 / 2.40 / 2.41. Să observăm că, tabelele de comenzi aferente instrucţiunilor ST *dest*,A şi ST *dest*,SP se diferenţiază la nivelul unei singure comenzi: comanda *PdA*, de la prima instrucţiune, devine *PdSP*, la a doua.

Tabelul de comenzi aferent instrucţiunii ST *dest*,FLAG este uşor diferit de cele aferente instrucţiunilor ST *dest*,A şi respectiv ST *dest*,SP. Aceasta se datorează faptului că registrul FLAG este emis direct pe RBUS şi nu prin intermediul ALU (figura 2.2).

Faza EX – instrucţiunea ST *dest,*A

I_1
PdA, SBUS, PdALU, TI_1 AD : *PmRG* AI : *WR* AX : *WR* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.39 Faza EX aferentă instrucţiunii **ST *dest*,A** – tabelul de comenzi

Faza EX – instrucţiunea ST *dest,*SP

I_1
PdSP, SBUS, PdALU, TI_1 AD : *PmRG* AI : *WR* AX : *WR* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.40 Faza EX aferentă instrucţiunii **ST *dest*,SP** – tabelul de comenzi

Faza EX – instrucţiunea ST *dest,*FLAG

I_1
PdFLAG, TI_1 AD : *PmRG* AI : *WR* AX : *WR* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.41 Faza EX aferentă instrucţiunii **ST *dest*,FLAG** – tabelul de comenzi

Să mai notăm că *fetch*-ul operandului din faza OF este inutil pentru instrucţiunile „*store*”. Totuşi, pentru modurile de adresare AI şi AX (cazul „*store*” în memorie), calculul adresei operandului care se face în cadrul fazei OF (blocurile (8) şi respectiv (9) din figura 2.19), este o operaţie utilă. Cu alte cuvinte, pentru modurile AI şi AX, faza OF conţine două operaţii succesive (fig 2.19):

1. calculul adresei operandului, în acord cu modul de adresare codificat în instrucţiune (blocurile (8) şi respectiv (9) din figura 2.19).

2. *fetch*-ul (citirea) operandului din memorie şi încărcarea acestuia în registrul M (blocul (10) din figura 2.19).

Prima operaţie este utilă şi necesară şi în cazul înstrucţiunilor „*store*” cu AI sau AX deoarece, în faza EX, vom face o scriere la această adresă. A doua operaţie este nenecesară dar nu deranjază. O fază OF mai rafinată şi desigur mai performantă ar elimina operaţia a doua atunci când se procesează o instrucţiune „*store*”.

I. Instrucţiunea JMP *adr*

Există următoarele forme uzuale (utile) pentru instrucţiunea de salt necondiţionat:

1. JMP 2000 - salt necondiţionat cu adresare imediată (AM). Constanta 2000 reprezintă adresa de salt (adresa instrucţiunii pe care se va executa saltul). Când programăm în asamblare utilizăm etichete pentru specificarea adreselor de salt, ca de exemplu: JMP ET1. Asamblorul va calcula adresa corespondentă pentru fiecare etichetă, va codifica această adresă în binar şi o va integra în codul instrucţiunii JMP. Există două moduri posibile de a proiecta instrucţiunile de salt cu adresare imediată:

   a). Salturi absolute. Formatul unei asemenea instrucţiuni este redat în figura 2.42 iar execuţia acesteia ar consta în:

   PC←ADRESA DE SALT

   b). Salturi relative (relative la adresa curentă din registrul PC). Formatul unei asemenea instrucţiuni este redat în figura 2.43 iar execuţia acesteia ar consta în:

   PC←PC+*OFFSET*

   OFFSET–ul reprezintă un număr cu semn care indică numărul de adrese sărit, faţă de o bază, baza fiind adresa curentă din PC. Prin urmare, dacă *OFFSET*–ul este pozitiv saltul se va face înainte iar dacă *OFFSET*–ul este negativ saltul se va face înapoi. În figura 2.44 se exemplifică un salt înainte (*OFFSET* pozitiv). *OFFSET*–ul va fi calculat de asamblor/compilator (în faza de asamblare/compilare a programului), va fi convertit în binar şi va fi inserat ca atare în codul instrucţiunii JMP. Relaţia de calcul implementată în asamblor/compilator, explicitată pe exemplul din figura 2.44, va fi:

$$OFFSET = ET1 - PC_{curent}$$

Cu alte cuvinte, *OFFSET*–ul va fi obţinut calculând diferenţa dintre adresa ţintă şi valoarea curentă din registrul PC.

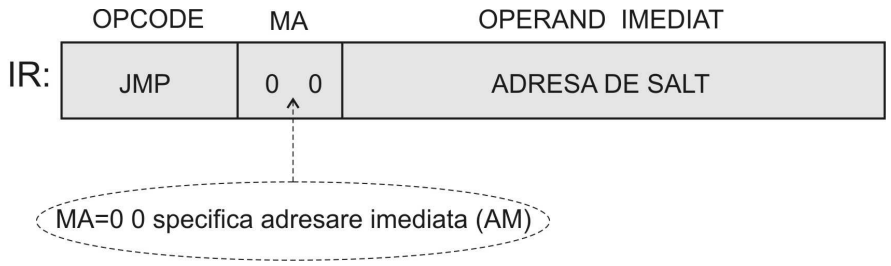


Fig. 2.42 Salturi absolute – formatul instrucţiunii

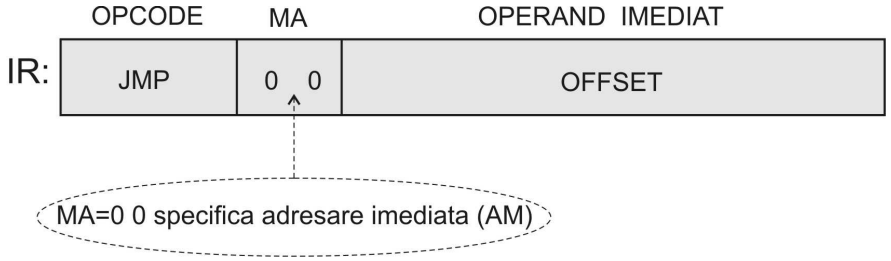


Fig. 2.43 Salturi relative – formatul instrucţiunii

În faza de execuţie a instrucţiunii JMP, adresa de salt (adresa ţintă) va fi calculată prin relaţia inversă:

$$ET1 = PC_{curent} + OFFSET$$

În faza EX a instrucţiunii JMP, în registrul PC se află deja adresa următoarei instrucţiuni din program (cea care urmează după JMP). Pentru această adresă am utilizat notaţia PC_{curent} (care semnifică valoarea existentă în registrul PC în momentul intrării în faza EX a instrucţiunii JMP).

Varianta „salturi relative” este larg utilizată deoarece permite relocarea programelor în memorie:

programul nu trebuie recompilat la fiecare reîncărcare a sa (care echivalează cu o mutare a sa) în memorie. Această variantă o vom implementa şi noi.

Varianta „salturi absolute” este deficitară deoarece revendică recompilarea/reasamblarea programului la fiecare reîncărcare (mutare) a acestuia în memorie; adresele de salt (adresele ţintă) sunt dependente valoric de adresa de start a programului şi prin urmare trebuie recalculate la fiecare reîncărcare/mutare a programului în memorie. Reîncărcarea/mutarea programelor în memorie este o operaţie frecventă în sistemele de operare *multitasking*.

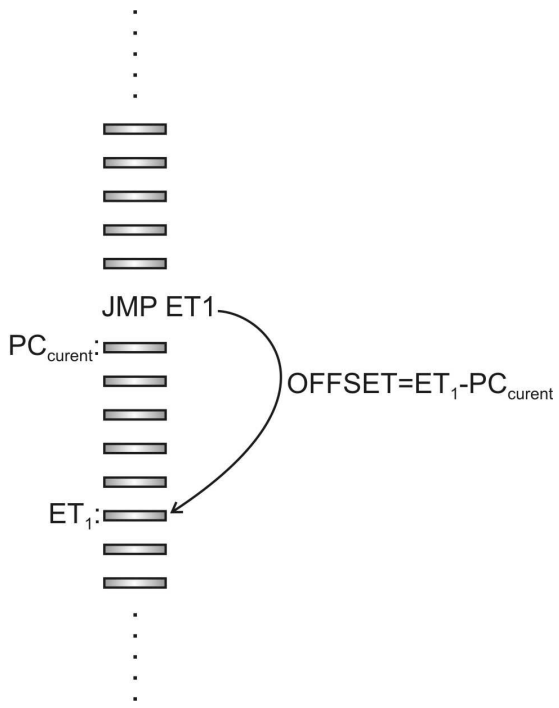


Fig. 2.44 Salt relativ înainte (*OFFSET* pozitiv)

2. JMP (R5) - salt necondiţionat cu adresare indirectă (AI). Execuţia unei asemenea instrucţiuni constă în:

$$PC \leftarrow R_5$$

Prin urmare, registrul general specificat în instrucţiune (în exemplul nostru R_5) furnizează adresa de salt (adresa ţintă).

3. JMP 6(R9) - salt necondiţionat cu adresare indexată (AX). Execuţia este similară cu cea din cazul AI, cu deosebirea că la calculul adresei de salt contribuie şi *INDEX*–ul codificat binar în codul instrucţiunii:

$$PC \leftarrow R_9+6$$

În exemplul nostru *INDEX*–ul are valoarea 6.

Observaţie:
JMP cu adresare directă (AD) reprezintă cod ilegal (caz detectat şi tratat în faza IF).

Dacă analizăm cele trei forme valide ale instrucţiunii JMP *adr* constatăm că avem doar două variante de obţinere a adresei de salt:

1. pe baza *OFFSET*–ului codificat în codul instrucţiunii, în cazul AM. În acest caz, pentru obţinerea adresei de salt în PC, se execută operaţia: PC←PC+*OFFSET*

2. pe baza registrului specificat, în cazul AI şi respectiv a registrului şi a *INDEX*–ului, în cazul AX. Analizând strict cazurile AI şi AX, putem constata că adresa de salt (care apare la instrucţiunile de salt) este identică cu adresa operandului (care apare la instrucţiunile non-salt). În faza OF (anterior parcursă), această adresă a fost încărcată în registrul ADR (blocurile (8) şi respectiv (9) din figura 2.19). Prin urmare, în faza EX a instrucţiunii JMP este suficientă operaţia PC←ADR (pentru încărcarea adresei de salt în registrul PC).

Organigrama fazei EX aferentă instrucţiunii de salt necondiţionat este redată în figura 2.45 iar tabelul de comenzi (optimizat) aferent acestei execuţii este redat în figura 2.46.

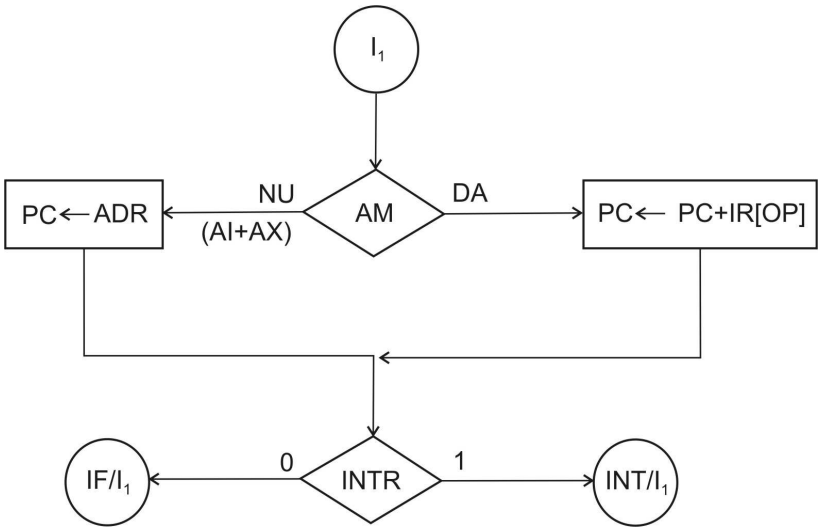


Fig 2.45 Organigrama fazei EX aferentă instrucţiunii **JMP *adr***

Faza EX – instrucţiunea JMP *adr*

I_1
PmPC, TI_1 AM : *PdIR[OP], PdPC, SUM, PdALU* $\overline{\text{AM}}$: *PdADR* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.46 Faza EX aferentă instrucţiunii **JMP *adr*** – tabelul de comenzi

IR[OP] reprezintă octetul inferior din IR, adică chiar OFFSET-ul. Menţiunea relativă la optimizarea fazei OF, făcută la instrucţiunile „*store*" (la sfârşitul paragrafului 2.7.1.H.), este valabilă şi aici.

J. Instrucţiunile de salt condiţionat

Fiecare dintre cele opt instrucţiuni de salt condiţionat (JC, JNC, JZ, JNZ, JP, JM, JV şi JNV) testează *flag*-ul de condiţii specificat (test pe adevărat sau pe fals) şi realizează saltul numai dacă condiţia testată este îndeplinită; în caz contrar saltul este inefectiv. Aceste instrucţiuni sunt similare cu instrucţiunea de salt necondiţionat (JMP *adr*), cu deosebirea că, încărcarea adresei de salt în PC se va face numai dacă condiţia testată este îndeplinită. În figura 2.47 este redat tabelul de comenzi generic aferent fazei de execuţie a celor opt instrucţiuni de salt condiţionat; singura deosebire faţă de tabelul aferent instrucţiunii de salt necondiţionat din figura 2.46, se referă la comanda *PmPC* care devine condiţionată la salturile condiţionate (figura 2.47). Prin urmare, expresia generică **Condiţie:** ***PmPC*** (care apare în tabelul 2.47) semnifică faptul că ***PmPC*** este comandă condiţionată. Altfel spus, ***Pm PC*** se activează numai dacă **Condiţie**=1.

Faza EX – Instrucţiuni de salt condiţionat

I_1
TI_1 AM : *PdIR[OP], PdPC, SUM, PdALU* $\overline{\text{AM}}$: *PdADR* **Condiţie**: *PmPC* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.47 Faza EX aferentă instrucţiunilor de salt condiţionat – tabelul de comenzi

Condiţia testată diferă de la o instrucţiune la alta. Tabelul 2.4 sintetizează condiţiile testate de cele opt instrucţiuni de salt condiţionat. Se observă că sunt testate de fapt (pe adevărat şi pe fals) cele patru *flag*-uri de condiţii.

Instrucţiune	Condiţie testată	
JC *adr*	C	(*Carry*=1 ?)
JNC *adr*	$\overline{C}$	(*Carry*=0 ?)
JZ *adr*	Z	(*Zero*=1 ?)
JNZ *adr*	$\overline{Z}$	(*Zero*=0 ?)
JP *adr*	$\overline{S}$	(*Sign*=0 ?)
JM *adr*	S	(*Sign*=1 ?)
JV *adr*	V	(*Overflow*=1 ?)
JNV *adr*	$\overline{V}$	(*Overflow*=0 ?)

Tabelul 2.4 Condiţiile testate de cele opt instrucţiuni de salt condiţionat

Pe baza condiţiilor din tabelul 2.4, tabelul de comenzi generic din figura 2.47 generează opt tabele de comenzi (câte unul pentru ficare instrucţiune de salt condiţionat). De exemplu, pentru a obţine tabelul de comenzi aferent instrucţiunii JC *adr*, condiţia generică „**Condiţie:**” (din figura 2.47), trebuie înlocuită cu condiţia concretă „**C:**” Analog se procedează pentru următoarele 7 instrucţiuni de salt condiţionat. La proiectarea blocului de generare a comenzilor (BGC), trebuie luate în considerare toate cele opt tabele concrete ce vor rezulta din tabelul generic !

K. Instrucţiunea CALL *adr*

Instrucţiunea CALL *adr* realizează apelul unei proceduri (procedura din limbajul de asamblare este denumită de regulă funcţie în limbajele de nivel înalt). Ultima instrucţiune a procedurii este întotdeauna RET (*return*). Prin RET se revine în programul apelant (figura 2.48).

Instrucţiunea CALL *adr* se execută în doi paşi:

1. PUSH PC ;salvează în stivă registrul PC. Altfel spus, salvează în stivă **adresa de** ;**revenire** în programul apelant. În faza EX a instrucţiunii CALL, ;registrul PC va conţine chiar adresa următoarei instrucţiuni, adică a celei ;care urmează după CALL (în apelant). Aceasta este tocmai adresa de ;revenire (notată **Adresa de revenire** în figura 2.48).

2. JMP *adr* ;salt necondiţounat la adresa *adr*.

În concluzie, instrucţiunea CALL *adr* este echivalentă cu două instrucţiuni (integrate într-una singură!): PUSH PC urmată de JMP *adr*.

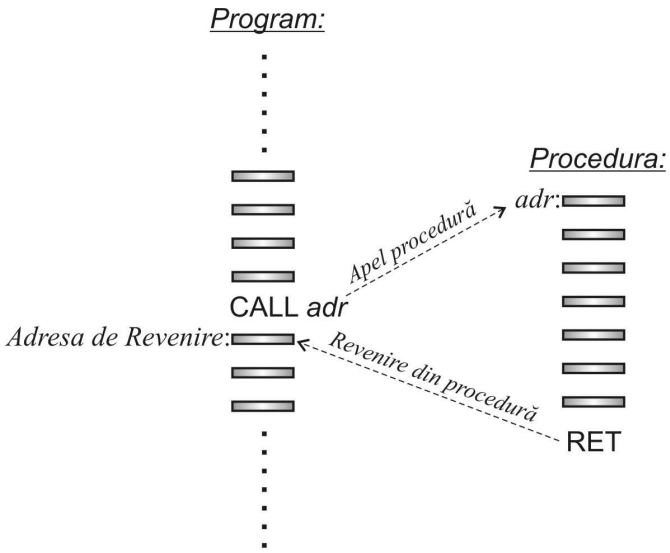


Fig 2.48 Mecanismul de apel şi respectiv de revenire din procedură

Tabelul de comenzi aferent fazei de execuţie a instrucţiunii CALL este redat în figura 2.49. Pe primele trei impulsuri de stare ($I_1 \div I_3$) se realizează operaţia *push PC* iar pe I_4 este implementat saltul necondiţionat la adresa *adr*.

Faza EX – instrucţiunea JMP *adr*

I_1	I_2	I_3	I_4
-2SP	*PdSP, SBUS, PdALU, PmADR*	*PdPC, DBUS, PdALU, WR*	*PdALU, PmPC, TI_1* AM : *PdIR[OP], PdPC, SUM* AI : *PdRG, DBUS* AX : *PdIR[IND], PdRG, SUM* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.49 Faza EX aferentă instrucţiunii **CALL adr** – tabelul de comenzi

Observaţii:

1. *Formele uzuale (utile) ale instrucţiunii JMP adr, prezentate la 2.7.1.I, sunt valabile şi implementate întocmai şi la instrucţiunea CALL adr.*

2. *Deşi am afirmat că pe impulsul I_4 din figura 2.44 este implementată practic faza EX a instrucţiunii JMP adr, constatăm că există diferenţe între coloana I_4 menţionată şi respectiv coloana I_1 din figura 2.46. Vom justifica, în cele ce urmează, aceste diferenţe. Instrucţiunile JMP adr şi CALL adr au trei moduri de adresare legale: AM, AI şi AX; JMP sau CALL cu AD reprezintă cod ilegal (condiţie detectată global şi tratată în faza IF). Dacă luăm în cosiderare doar cazurile AI şi AX, putem constata că adresa de salt (respectiv de call), care apare doar la instrucţiunile JMP (respectiv CALL), este identică cu adresa operandului, care apare la instrucţiunile non-salt (respectiv non-call). În faza OF (anterior parcursă) această adresă a fost încărcată în registrul ADR (blocurile (8) şi respectiv (9) din figura 2.19). Prin urmare, în faza EX aferentă instrucţiunilor JMP şi CALL ar fi suficientă operaţia PC←ADR pentru a încărca adresa de salt (respectiv call) în registrul PC. Acest raţionament a fost aplicat întocmai la instrucţiunea JMP (figurile 2.45 şi 2.46), unde am considerat că $AD \vee AX = \overline{AM}$ (operatorul $\vee$ semnifică funcţia logică SAU). Această minimizare realizată la nivelul tabelului de comenzi din figura 2.46 va conduce evident la simplificarea structurii hardware aferentă BGC-ului.*

   Raţionamentul nu poate fi aplicat şi la instrucţiunea CALL deoarece aici avem de-a face cu cei doi paşi succesivi:

   1. push PC (operaţie implementată pe $I_1 \div I_3$, în figura 2.49)

   2. JMP adr (operaţie implementată pe I_4, în figura 2.49)

   În prima operaţie (push PC), registrul ADR este încărcat (pe I_2) şi utilizat (pe I_3) pentru adresarea locaţiei din vârful stivei. Prin urmare, adresa de call, necesară în a doua operaţie (pe I_4), se pierde. Această adresă trebuie reîncărcată în ADR prin operaţii similare celor din faza OF (blocurile (8) şi respectiv (9) din figura 2.19).

3. *Dacă instrucţiunea CALL salvează adresa de revenire în stivă, printr-o operaţie push PC, înseamnă că instrucţiunea RET, de la sfârşitul procedurii, va trebui să realizeze operaţia pop PC. În acest mod, instrucţiunea RET va reîncărca în PC adresa de revenire în programul apelant. Deci, instrucţiunea RET este de fapt instrucţiunea POP PC.*

2.7.2 Faza EX pentru instrucţiunile din clasa ID

A. Instrucţiunile INC A, DEC A şi NEG A

Cele trei instrucţiuni operează cu registrul A (operand implicit); acţiunea lor (definită în tabelul 2.3) poate fi sintetizată astfel:

INC A ;A←A+1 (incrementare A) şi setare *flag*-uri (ca la adunare).

DEC A ;A←A-1 (decrementare A) şi setare *flag*-uri (ca la scădere).

NEG A ;A← $\overline{A}$ (complementare A) şi setare *flag*-uri (ca la instrucţiunile logice).

Primele două instrucţiuni sunt instrucţiuni aritmetice (de adunare şi respectiv scădere) iar ultima este instrucţiune logică; setarea *flag*-urilor de condiţii trebuie făcută în acord cu tipul instrucţiunii (la instrucţiunea INC A ca la adunare, la instrucţiunea DEC A ca la scădere, iar la instrucţiunea NEG A ca la instrucţiunile logice). Tabelele de comenzi aferente fazei de execuţie a celor trei instrucţiuni sunt redate în figurile 2.50, 2.51 şi 2.52.

Faza EX – instrucţiunea INC A

I_1
PdA, Pd0$_d$, C$_{in}$, SUM, PdALU, PmA, TI$_1$
C_{out} : *A(1)C*
$\overline{C}_{out}$: *A(0)C*
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
DCR_{ADD} : *A(1)V*
$\overline{DCR}_{ADD}$: *A(0)V*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.50 Faza EX aferentă instrucţiunii **INC A** – tabelul de comenzi

Comanda *Pd0$_d$*, utilizată la instrucţiunea INC A, emite constanta zero pe DBUS. Modul în care este impementată în *hardware* este ilustrat în figura 2.53.

Comanda *Pd-1$_d$*, utilizată la instrucţiunea DEC A, emite constanta -1 (minus unu) pe DBUS. Modul în care este impementată în *hardware* este ilustrat în aceeaşi figură 2.53.

Comanda C_{in} aplică un unu logic pe intrarea C_{in} *(carry input)* a sumatorului din componenţa ALU iar comanda *SUM* validează acest sumator. Astfel, la instrucţiunea INC A, se face operaţia A←A+C_{in}=A+1.

La instrucţiunea DEC A, comenzile *PdA*, *Pd-1$_d$ şi SUM* (care acţionează desigur concurent) realizează operaţia A←A+(-1)=A-1.

Comanda $\overline{SBUS}$ face parte din setul de comenzi dedicate unităţii ALU. Aceasta activează un bloc format din 16 inversoare (din componenţa ALU), determinând inversarea, bit cu bit, a operandului preluat de pe SBUS. În acest fel, ALU inversează, bit cu bit, operandul preluat de pe SBUS (la instrucţiunea NEG A).

Faza EX – instrucţiunea DEC A

I_1
PdA, Pd-1$_d$, SUM, PdALU, PmA, TI$_1$
C_{out} : *A(1)C*
$\overline{C}_{out}$: *A(0)C*
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
DCR_{SUB} : *A(1)V*
$\overline{DCR}_{SUB}$: *A(0)V*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.51 Faza EX aferentă instrucţiunii **DEC A** – tabelul de comenzi

Faza EX – instrucţiunea NEG A

I_1
PdA, $\overline{SBUS}$ *, PdALU, PmA, TI$_1$*
Z_R : *A(1)Z*
$\overline{Z}_R$: *A(0)Z*
S_R : *A(1)S*
$\overline{S}_R$: *A(0)S*
$\overline{INTR}$: *TIF*
INTR : *TINT*

Fig 2.52 Faza EX aferentă instrucţiunii **NEG A** – tabelul de comenzi

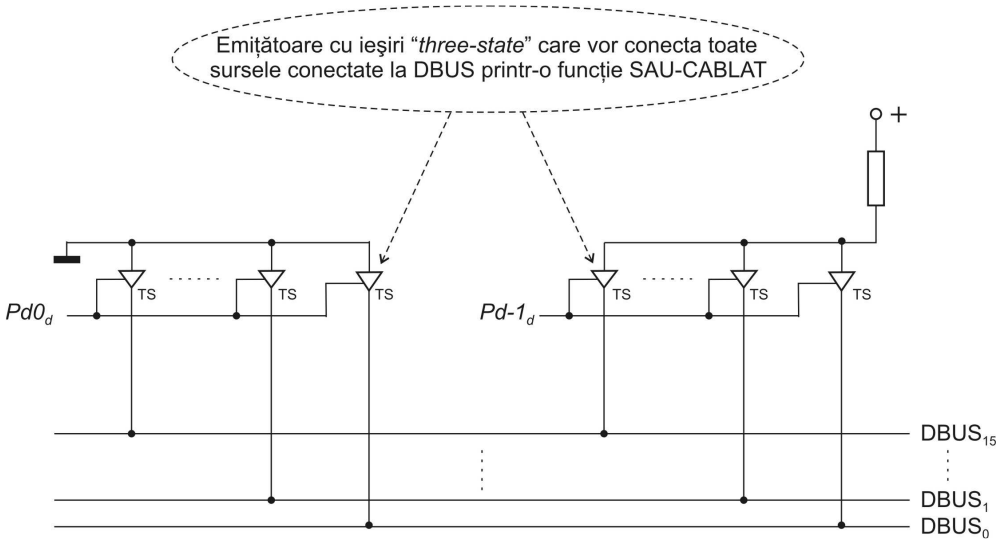


Fig 2.53 Implementarea comenzilor $Pd0_d$ şi $Pd\text{-}1_d$ în *hardware*

B. Instrucţiunile PUSH A şi PUSH FLAG

Instrucţiunile PUSH A şi PUSH FLAG salvează (memorează) conţinutul registrului A, respectiv FLAG, în stivă. Sunt similare cu instrucţiunea PUSH *reg* din clasa IOP. Deosebirea este că la PUSH A şi PUSH FLAG nu se mai parcurge faza OF deoarece sunt instrucţiuni din clasa ID (operandul salvat în stivă este referit implicit prin *OPCODE*-ul instrucţiunii şi nu prin câmpurile MA şi ADR care apar la PUSH *reg* şi care referă în mod explicit operandul *reg*). Această deosebire nu este însă una de funcţionalitate. Prin urmare, instrucţiunile PUSH A şi PUSH FLAG au o fază de execuţie similară celei prezentate la instrucţiunea PUSH *reg* (vezi figura 2.37 cu explicaţiile aferente). Tabelele de comenzi aferente fazei de execuţie a celor două instrucţiuni sunt redate în figurile 2.54 şi 2.55.

Coloanele I_1 şi I_2 din figura 2.55 sunt identice cu cele din figura 2.54 şi mai departe, identice cu cele din figura 2.37.

În afară de punctul interuptibil, pe impulsul I_3, sunt activate comenzi care realizează următoarele operaţii:

1. La instrucţiunea PUSH A:

   - *PdA*, *SBUS* şi *PdALU* care, activate simultan, emit pe RBUS conţinutul registrului A.

   - *WR* care scrie în memorie (stivă) datele preluate de pe MBUS (RBUS); vezi figura 2.36 şi explicaţiile aferente.

2. La instrucţiunea PUSH FLAG:

   - *PdFLAG* care emite pe RBUS (direct şi nu prin intermediul ALU) conţinutul registrului FLAG.
   - *WR* care scrie în memorie (stivă) datele preluate de pe MBUS (RBUS).

Faza EX – instrucţiunea PUSH A

I_1	I_2	I_3
-2SP	*PdSP, SBUS, PdALU, PmADR*	*PdA, SBUS, PdALU, WR, TI_1* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.54 Faza EX aferentă instrucţiunii **PUSH A** – tabelul de comenzi

Faza EX – instrucţiunea PUSH FLAG

I_1	I_2	I_3
-2SP	*PdSP, SBUS, PdALU, PmADR*	*PdFLAG, WR, TI_1* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.55 Faza EX aferentă instrucţiunii **PUSH FLAG** – tabelul de comenzi

C. Instrucţiunile POP A şi POP FLAG

Instrucţiunile POP A şi POP FLAG restaurează conţinutul registrului A, respectiv FLAG, din stivă. Sunt similare cu instrucţiunea POP *reg* din clasa IOP. Tabelele de comenzi aferente fazei de execuţie a celor două instrucţiuni sunt redate în figurile 2.56 şi 2.57.

Faza EX – instrucţiunea POP A

I_1	I_2
PdSP, SBUS, PdALU, PmADR	*RD, PdMEM, PmA, +2SP, TI_1* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.56 Faza EX aferentă instrucţiunii **POP A** – tabelul de comenzi

Faza EX – instrucţiunea POP FLAG

I_1	I_2
PdSP, SBUS, PdALU, PmADR	*RD, PdMEM, PmFLAG, +2SP, TI$_1$* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.57 Faza EX aferentă instrucţiunii **POP FLAG** – tabelul de comenzi

Să observăm că tabelele din figurile 2.56, 2.57 şi 2.38 sunt aproape identice. Acestea diferă printr-o singură comandă, de pe colana I_2:

- *PmRG*, în cazul instrucţiunii POP *reg* (fig. 2.38), încarcă datele citite din stivă în registrul general *reg* specificat în instrucţiune (instrucţiunea are de fapt forma POP R_i).

- *PmA*, în cazul instrucţiunii POP A (fig. 2.56), încarcă datele citite din stivă în registrul A.

- *PmFLAG*, în cazul instrucţiunii POP FLAG (fig. 2.57), încarcă datele citite din stivă în registrul FLAG.

D. Instrucţiunile de deplasare şi rotire

Instrucţiunile de deplasare şi rotire au fost introduse la 2.2.2. În arhitecturile bazate pe acumulator (cum este procesorul didactic) aceste instrucţiuni operează de regulă asupra acumulatorului. Prin urmare, în aceste arhitecturi, registrul A devine un registru de deplasare stînga-dreapta (fig. 2.58) care, pe lângă comenzile consacrate *PdA* şi *PmA*, va mai executa două comenzi de deplasare:

- *DrA* –deplasare la dreapta cu o poziţie (un bit).

- *StA* –deplasare la stânga cu o poziţie (un bit).

După cum indică figura 2.58, pentru efectuarea operaţiilor de deplasare stânga-dreapta, sunt necesare şi două intrări seriale în registrul A:

- IS_{15} –intrare serială în rangul 15 (cel mai semnificativ), necesară pentru operaţiile de deplasare (rotire) la dreapta.

- IS_0 –intrare serială în rangul zero (cel mai puţin semnificativ), necesară pentru operaţiile de deplasare (rotire) la stânga.

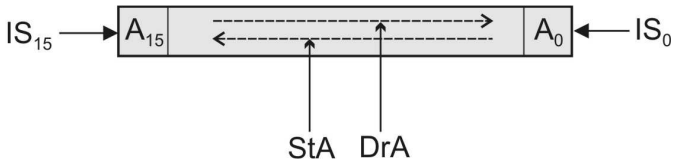


Fig 2.58 Comenzile de deplasare şi intrările seriale în registrul A

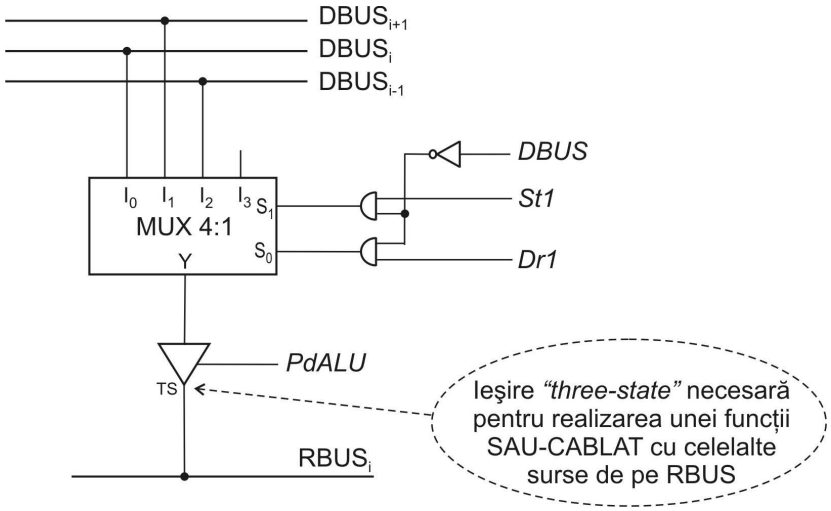
Observaţii:

1. *În arhitecturile bazate strict pe registre generale (fără acumulator), instrucţiunile de deplasare şi rotire operează asupra registrelor generale (de exemplu, instrucţiunea ASL A devine ASL R_i, unde R_i poate fi orice registru general). În aceste arhitecturi, fără acumulator, implementarea în hardware a operaţiilor de deplasare şi rotire se face de regulă la nivelul unităţii ALU. Unitatea ALU va fi dotată cu un bloc combinaţional adiţional (blocul SHIFT) care va realiza operaţiile de deplasare şi rotire. Deplasarea (rotirea) se poate realiza într-o singură perioadă de tact, astfel: conţinutul registrului R_i este emis pe DBUS (cu ajutorul comenzii PdRG), se aplică pe intrarea unităţii ALU (blocului SHIFT), blocul SHIFT realizează deplasarea (rotirea) solicitată emiţând pe RBUS rezultatul deplasării (rotirii), de unde rezultatul va fi reîncărcat în acelaşi registru (cu ajutorul comenzii PmRG). Blocul SHIFT va fi format în principiu dintr-o reţea de multiplexoare. De exemplu, cu multiplexorul 4:1 din figura 2.59, controlat de cele două intrări de selecţie S_1 şi S_0, se pot implementa deplasările din tabelul 2.5. Să observăm totuşi că multiplexorul din figura 2.59 implementează operaţiile din tabelul 2.5, la nivelul unui singur bit (bitul $RBUS_i$). Pentru a extinde operaţiile respective la nivelul tuturor celor 16 biţi ($RBUS_{15\div0}$), structura din figura 2.59 trebuie multiplicată de 16 ori.*
   Să mai menţionăm că, în arhitecturile bazate strict pe registre generale, instrucţiunile de deplasare şi rotire ar putea opera şi cu operanzi locaţi în memorie. Majoritatea procesoarelor CISC execută şi instrucţiuni de deplasare (rotire) care operează cu astfel de operanzi, ca de exemplu:

   ASR (R_i) –*adresare indirectă*

   ROL k(R_i) –*adresare indexată*

2. *Procesoarele CISC comerciale execută de regulă şi instrucţiuni de deplasare (rotire) stânga/dreapta cu mai multe poziţii. Structura blocului SHIFT, în aceste situaţii, este similară celei sintetizate în figura 2.59, doar că se utilizează multiplexoare mai complexe (cu mai multe intrări). Pentru a adăuga deplasări stânga/dreapta cu doi biţi de exemplu, trebuie implementate încă două intrări la multiplexorul din figura 2.59, la care se vor conecta liniile $DBUS_{i-2}$ şi respectiv, $DBUS_{i+2}$.*



DBUS -comandă care trece nemodificat prin ALU operandul preluat de pe DBUS
St1 -comandă de deplasare la stânga cu o poziţie
Dr1 -comandă de deplasare la dreapta cu o poziţie

Fig 2.59 Structura de principiu a blocului SHIFT

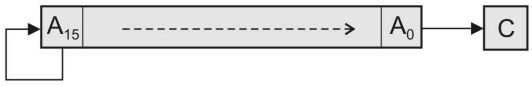
Comenzi			Operaţie efectuată	Tip deplasare
DBUS	***St1***	***Dr1***		
1	×	×	RBUS$_i$←DBUS$_i$	Nici o deplasare (*DBUS*=1)
0	0	1	RBUS$_i$←DBUS$_{i+1}$	Deplasare la dreapta cu o poziţie (*Dr1*=1)
0	1	0	RBUS$_i$←DBUS$_{i-1}$	Deplasare la stânga cu o poziţie (*St1*=1)
0	1	1	—	Operaţie nedefinită (dezvoltări viitoare)

Tabelul 2.5 Cele trei operaţii implementate cu blocul *SHIFT* din figura 2.59

În cadrul procesorului didactic, după cum am menţionat anterior, vom implementa deplasările şi rotirile pe registrul A. Aceste operaţii de deplasare şi rotire vor afecta registrul A şi *flag*-ul C (*Carry*). Ca regulă de principiu, la deplasări, bitul care iese serial din registru se introduce întotdeauna în *flag*-ul *Carry*. Schemele de principiu după care se realizează instrucţiunile de deplasare şi rotire sunt redate în figura 2.60.

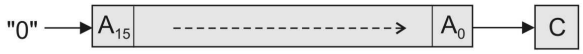
$A_i \leftarrow A_{i-1}$; i=1,2,...,15
$C \leftarrow A_{15}$
$A_0 \leftarrow 0$

a) ASL-*Arithmetic Shift Left* (deplasare aritmetică la stânga); este identică cu LSL-*Logical Shift Left* (deplasare logică la stânga)



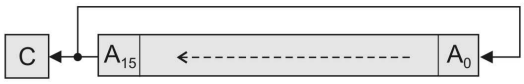
$A_{i-1} \leftarrow A_i$; i=1,2,...,15
$C \leftarrow A_0$
$A_{15} \leftarrow A_{15}$

b) ASR-*arithmetic shift right* (deplasare aritmetică la dreapta)



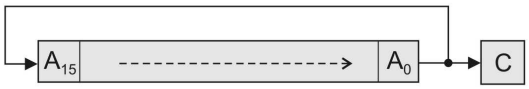
$A_{i-1} \leftarrow A_i$; i=1,2,...,15
$C \leftarrow A_0$
$A_{15} \leftarrow 0$

c) LSR-*Logical Shift Right* (deplasare logică la dreapta)



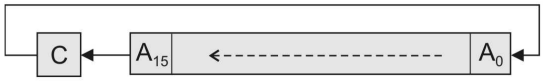
$A_i \leftarrow A_{i-1}$; i=1,2,...,15
$C \leftarrow A_{15}$
$A_0 \leftarrow A_{15}$

d) ROL-*ROtate Left* (rotire la stânga)



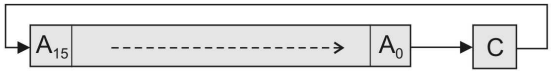
$A_{i-1} \leftarrow A_i$; i=1,2,...,15
$C \leftarrow A_0$
$A_{15} \leftarrow A_0$

e) ROR-*ROtate Right* (rotire la dreapta)



$A_i \leftarrow A_{i-1}$; i=1,2,...,15
$C \leftarrow A_{15}$
$A_0 \leftarrow C$

f) RLC-*Rotate Left through Carry* (rotire la stânga cu *Carry*)



$A_{i-1} \leftarrow A_i$; i=1,2,...,15
$C \leftarrow A_0$
$A_{15} \leftarrow C$

g) RRC-*Rotate Right through Carry* (rotire la dreapta cu *Carry*)

Fig 2.60 Acţiunea instrucţiunilor de deplasare şi rotire asupra registrului A şi asupra *flag*-ului C

Tot în figura 2.60 sunt redate şi expresiile matematice care definesc fiecare operaţie de deplasare şi respectiv rotire.
Sintetizând, putem concluziona că, în funcţie de sens, deplasările şi rotirile sunt de două tipuri:

- deplasări (respectiv rotiri) la stânga
- deplasări (respectiv rotiri) la dreapta

În funcţie de obiectivul urmărit, deplasările (nu şi rotirile!) sunt tot de două tipuri:

- deplasări aritmetice. Aceste deplasări tratează cuvântul de 16 biţi supus deplasării ca pe o entitate aritmetică (număr reprezentat în cod complementar, cu bit de semn şi respectiv parte de mărime). Prin urmare, deplasările aritmetice vizează efectuarea unei operaţii aritmetice asupra operandului vizat (deplasarea aritmetică la stânga realizează de fapt o înmulţire cu 2 iar deplasarea aritmetică la dreapta realizează o împărţire la 2). Acesta este argumentul intuitiv pe care îl putem invoca de exemplu, pentru explicarea operaţiei $A_{15} \leftarrow A_{15}$ (fig 2.60.b).) la deplasarea aritmetică la dreapta (instrucţiunea ASR). Prin $A_{15} \leftarrow A_{15}$ se realizează de fapt conservarea semnului. Asta e, cu ASR urmărim ca din operandul X să obţinem rezultatul X/2. Ori dacă X este pozitiv (respectiv negativ) şi X/2, rezultat în urma deplasării, va fi pozitiv (respectiv negativ). Rezultă că, deplasarea în discuţie trebuie să conserve bitul de semn (bitul A_{15}). Acest argument intuitiv nu este şi unul riguros din punct de vedere matematic. Relaţiile care definesc deplasările aritmetice la stânga şi respectiv dreapta pot fi demonstrate matematic, făcând uz de ecuaţiile matematice care definesc codul complementar (complementul faţă de doi). Nu ne-am propus să facem aici aceste demonstraţii; ne limităm la a prezenta doar argumente intuitive.

- deplasări logice. Aceste deplasări tratează cuvântul de 16 biţi supus deplasării ca pe o entitate logică şi nu ca pe una aritmetică. Regulile de bază aplicate la deplasările logice sunt:

  -cei 16 biţi ai registrului implicat se deplasează cu o poziţie în sensul specificat

  -bitul care iese serial din registru (la un capăt al acestuia) se introduce în *flag*-ul C

  -pe poziţia eliberată din registru (la celălalt capăt al acestuia) se introduce un ”0”

În funcţie de modul de implicare a *flag*-ului *Carry*, rotirile (nu şi deplasările!) sunt de două tipuri:

- rotiri simple (fără *Carry*). Astfel de rotiri realizează instrucţiunile ROL şi ROR, la care *flag*-ul *Carry* nu este inclus în circuitul de rotire.
- Rotiri cu *Carry* (*flag*-ul *Carry* inserat în circultul de rotire). Astfel de rotiri realizează instrucţiunile RLC şi RRC.

Tabelele de comenzi pentru faza de execuţie aferentă instrucţiunilor de deplasare şi rotire sunt sintetizate în figura 2.61. Aceste tabele au fost proiectate în acord cu schemele de principiu din figura 2.60.

Faza EX – instrucţiuni ASL şi LSL

I_1	
StA, TI_1	(IS_0=0)
$\overline{A_{15}}$: *A(0) C*	
A_{15} : *A(1) C*	
$\overline{INTR}$: *TIF*	
INTR : *TINT*	

Faza EX – instrucţiunea ROL

I_1	
StA, TI_1	(IS_0=A_{15})
$\overline{A_{15}}$: *A(0) C*	
A_{15} : *A(1) C*	
$\overline{INTR}$: *TIF*	
INTR : *TINT*	

Faza EX – instrucţiunea ASR

I_1	
DrA, TI_1	(IS_{15}=A_{15})
$\overline{A_0}$: *A(0) C*	
A_0 : *A(1) C*	
$\overline{INTR}$: *TIF*	
INTR : *TINT*	

Faza EX – instrucţiunea ROR

I_1	
DrA, TI_1	(IS_{15}=A_0)
$\overline{A_0}$: *A(0) C*	
A_0 : *A(1) C*	
$\overline{INTR}$: *TIF*	
INTR : *TINT*	

Faza EX – instrucţiunea LSR

I_1	
DrA, TI_1	(IS_{15}=0)
$\overline{A_0}$: *A(0) C*	
A_0 : *A(1) C*	
$\overline{INTR}$: *TIF*	
INTR : *TINT*	

Faza EX – instrucţiunea RLC

I_1	
StA, TI_1	(IS_0=C)
$\overline{A_{15}}$: *A(0) C*	
A_{15} : *A(1) C*	
$\overline{INTR}$: *TIF*	
INTR : *TINT*	

Faza EX – instrucţiunea RRC

I_1	
DrA, TI_1	(IS_{15}=C)
$\overline{A_0}$: *A(0) C*	
A_0 : *A(1) C*	
$\overline{INTR}$: *TIF*	
INTR : *TINT*	

Fig 2.61 Faza EX aferentă instrucţiunilor de deplasare şi rotire (tabele de comenzi)

Să notăm că deplasarea aritmetică la stânga (instrucţiunea ASL) este identică cu deplasarea logică la stânga (instrucţiunea LSL), după cum indică şi figura 2.60.a). Este motivul pentru care am rezervat o singură instrucţiune maşină (un singur *OPCODE*) pentru ambele instrucţiuni, pe care le regăsim de altfel distincte la nivelul limbajului de asamblare. Cu alte cuvinte, la nivelul limbajului de asamblare, asamblorul va recunoaşte două instrucţiuni diferite (ASL şi respectiv LSL), dar pentru ambele mnemonici va genera aceeaşi instrucţiune maşină. Această decizie vizează desigur simplificarea *hardware*-ului.

Pentru fiecare instrucţiune, în tabelul de comenzi, am specificat (între paranteze) şi modul în care trebuie acţionată intrarea serială în registrul A. Aceste ecuaţii scrise între paranteze vor fi implementate direct în *hardware* (fig. 2.62) şi nu prin comenzi logice generate de către BGC.

Comenzile *A(1)C* („adu la unu *carry*") şi respectiv *A(0)C* („adu la zero *carry*") sunt comenzi de setare şi respectiv de resetare a *flag*-ului *Carry*. Aceste comenzi le regăsim şi la instrucţiunile aritmetice (ADD A,*src*; SUB A,*src*; CMP A,*src*; INC *src*; DEC *src* etc.), unde realizează poziţionarea *flag*-ului C în acord cu rezultatul obţinut în instrucţiune. Acţiunea acestor comenzi în *hardware* este ilustrată în figura 2.62. Pe de altă parte, *flag*-ul C (bitul 3 din registrul FLAG) trebuie să poată fi încărcat de pe RBUS (cu starea liniei $RBUS_3$), la activarea comenzii *PmFLAG*. Această acţiune o regăsim, de exemplu în fazele de execuţie aferente instrucţiunilor POP FLAG şi respectiv IRET (fig. 2.57 şi 2.70). În figura 2.62 regăsim şi acţiunea comenzii *PmFLAG*.

Variabilele/semnalele ce trebuie aplicate pe intrările seriale (IS_0 şi respectiv IS_{15}) ale registrului A au fost evidenţiate atât în schemele de principiu din figura 2.60 cât şi în tabelele de comenzi din figura 2.61 (aici între paranteze). Aceste variabile/semnale sunt sintetizate în tabelul 2.6. În momentul activării comenzii de deplasare (*DrA* sau *StA*, după caz), intrările seriale în registrul A trebuie acţionate aşa cum se indică în coloana trei a tabelului 2.6. Implementarea în *hardware* a acestor acţiuni este sintetizată în figura 2.62.

Instrucţiune	Comandă de deplasare activată	Variabila aplicată pe intrarea serială
ASL (LSL)	*StA*	$IS_0 \leftarrow 0$
ASR	*DrA*	$IS_{15} \leftarrow A_{15}$
LSR	*DrA*	$IS_{15} \leftarrow 0$
ROL	*StA*	$IS_0 \leftarrow A_{15}$
ROR	*DrA*	$IS_{15} \leftarrow A_0$
RLC	*StA*	$IS_0 \leftarrow C$
RRC	*DrA*	$IS_{15} \leftarrow C$

Tabelul 2.6 Variabilele aplicate pe intrările seriale aferente registrului A

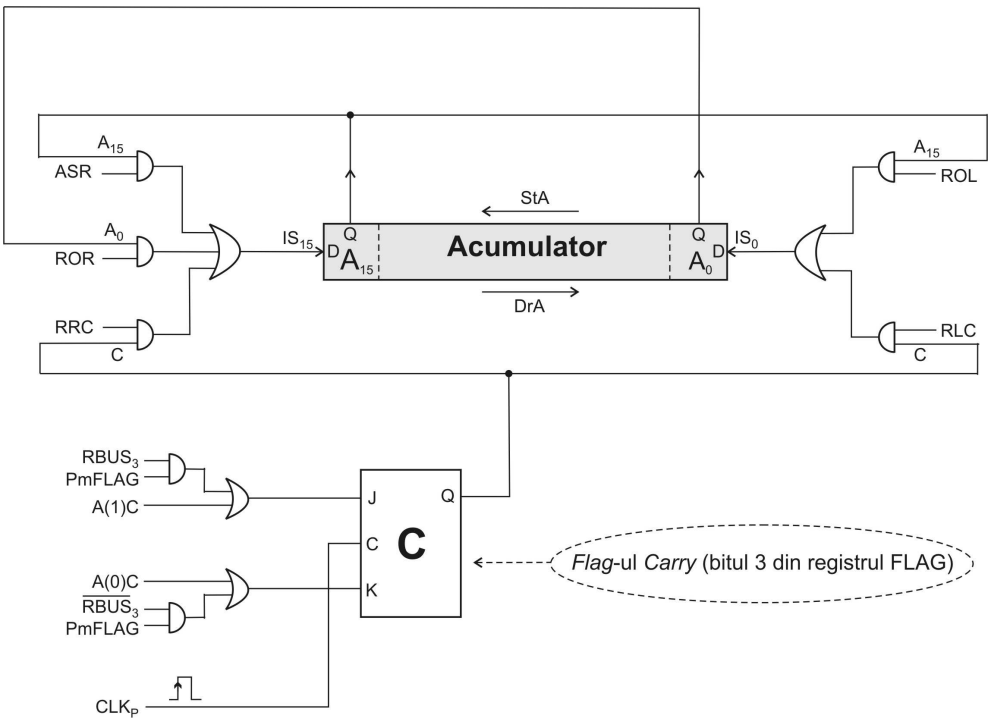


Fig 2.62 Acţiunea instrucţiunilor de deplasare şi rotire asupra intrărilor seriale aferente registrului A şi asupra *flag*-ului C

E. Instrucţiunile NOP şi HALT

Instrucţiunea NOP (*No OPeration*) va conţine doar punctul interuptibil (nu face nimic). Este utilă, de exemplu, la implementarea rutinelor de *delay* (întârziere), unde un anumit număr de NOP-uri în buclă pot realiza întârzierea dorită. Compilatoarele moderne (cu reorganizare) apelează la instrucţiunea NOP, pentru implementarea unor tehnici de optimizare (în faza de optimizare statică), care vizează minimizarea timpului global de execuţie al fiecărui program; este vorba desigur de compilatoarele dedicate procesoarelor cu paralelism la nivelul instrucţiunii (arhitecturi *pipeline* şi respectiv superscalare).
Tabelul de comenzi aferent fazei de execuţie a instrucţiunii NOP este redat în figura 2.63.

Faza EX – instrucţiunea NOP

I_1
TI_1 $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.63 Faza EX aferentă instrucţiunii **NOP** – tabelul de comenzi

Instrucţiunea HALT realizează oprirea procesorului, prin blocarea tactului acestuia. Acastă acţiune se poate implementa cu ajutorul unui bistabil de pornire-oprire (BPO), ca în figura 2.64.

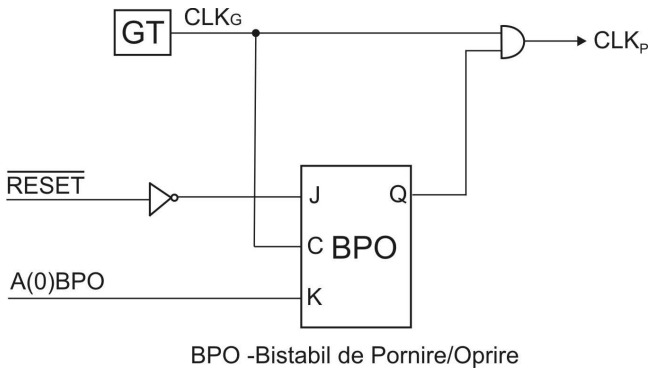


Fig 2.64 Validarea/blocarea semnalului de tact aferent procesorului cu ajutorul bistabilului BPO

Semnalul CLK_G este un tren de impulsuri (semnalul de tact), generat de către generatorul de tact (GT), atâta timp cât sistemul este sub tensiune (alimentat cu energie electrică). Acest tren de impulsuri CLK_G devine semnal de tact pentru procesor (CLK_P) numai dacă bistabilul BPO este setat. Bistabilul BPO este setat prin activarea semnalului $\overline{RESET}$, generat fie din sursa de alimentare (la punerea sub tensiune a sistemului), fie prin acţionarea butonului de RESET de către operator (dacă sistemul este deja sub tensiune).

Resetarea bistabilului BPO se realizează prin activarea comenzii *A(0)BPO* („adu la zero BPO”). Primul impuls CLK_G (primul front urcător) care survine după activarea comenzii *A(0)BPO* resetează bistabilul, blocând astfel generarea impulsurilor CLK_P. Sistemul poate fi deblocat (repornit) doar setând bistabilul BPO (prin acţionarea butonului RESET). Tabelul de comenzi aferent fazei de execuţie a instrucţiunii HALT este redat în figura 2.65.

Faza EX – instrucţiunea HALT

I_1
TIF, TI_1, A(0)BPO

Fig 2.65 Faza EX aferentă instrucţiunii **HALT** – tabelul de comenzi

Din moment ce instrucţiunea HALT realizează oprirea procesorului, punctul interuptibil nu mai are sens şi prin urmare n-a mai fost implementat în figura 2.65. Comenzile *TIF* şi *TI_1*

realizează iniţializarea GF şi respectiv GIS. Aceasta înseamnă că, la repornire, procesorul startează cu *fetch*-ul instrucţiuni de la adresa furnizată de registrul PC. Printre altele, semnalul $\overline{\text{RESET}}$ încarcă şi o adresă (o constantă) în registrul PC. Aceasta trebuie să fie o adresă de ROM (la IBM-PC o adresă de BIOS). Este adresa primei instrucţiuni pe care sistemul o execută după iniţializarea prin comanda $\overline{\text{RESET}}$; $\overline{\text{RESET}}$ la punerea sub tensiune (când în RAM nu are încă încărcat nici-un program) sau $\overline{\text{RESET}}$ activat de către operator (prin acţionarea butonului RESET).

Să notăm că instrucţiunea HALT este o instrucţiune „protejată”. Ea nu este executată decât în mod *supervisor* (denumit uneori şi *kernel*). Sistemele de operare actuale sunt sisteme protejate, în care sunt implementate cel puţin două nivele de privilegiu, nivelul *supervisor* şi nivelul *user*. Pentru gestiunea celor două nivele de privilegiu, procesorul pune la dispoziţia sistemului de operare anumite funcţii (implementate în *hardware*). Nivelul de privilegiu se setează cu ajutorul unui *flag* plasat într-un registru de control al procesorului (la anumite procesoare chiar în registrul FLAG). Sistemul de operare lucrează în mod *supervisor*; toate programele din componenţa sistemului de operare sunt executate în mod *supervisor*. Toate programele de aplicaţii sunt rulate în mod *user*. Sub sistemele de operare *multitasking*, la un moment dat, sunt active (procesate concurent) mai multe *task*-uri (programe). La nivelul sistemului de operare se formează aşa zisul lanţ de *task*-uri active. Aceste *task*-uri sunt baleiate periodic, fiecare primind un slot de **timp UC** (timp procesor). Când slotul acordat unui *task* se încheie, prin întreruperea de timp real, se declanşează o operaţie de comutare a *task*-urilor, trecându-se la *task*-ul următor. Printr-o baleiere (de exemplu circulară) a tuturor *task*-urilor active şi prin comutarea periodică de la *task*-ul curent la *task*-ul următor, utilizatorul are impresia unei execuţii concurente a tuturor *task*-urilor active. Lanţul de *task*-uri active conţine şi *task*-uri de sistem (programe din sistemul de operare) şi *task*-uri de aplicaţii. La intrarea într-un nou *task* (prin operaţia de comutare) se va instala automat şi nivelul de privilegiu asociat noului *task* (*user* sau *supervisor*). În mod *supervisor*, procesorul execută toate instrucţiunile, inclusiv instrucţiunile protejate (cum este şi instrucţiunea HALT). În mod *user*, procesorul nu execută instrucţiunile protejate. O instrucţiune HALT, plasată într-un progtam de aplicaţie, va fi ignorată (transformată în NOP). Este firesc ca sistemul de operare să se protejeze, inclusiv de tentativele de oprire a calculatorului pe care le-ar putea iniţia programele de aplicaţii.

F. Instrucţiunile EI şi DI

Instrucţiunea EI validează întreruperile (prin setarea *flag*-ului BVI) iar instrucţiunea DI invalidează întreruperile (prin resetarea *flag*-ului BVI). Atâta timp cât bistabilul BVI este resetat (figura 2.68) întreruperile externe sunt mascate. Întreruperi mascate înseamnă întreruperi nerecepţionate şi prin urmare netratate de către unitatea de control a procesorului (de către blocul BGC din figura 2.2).

La procesorul didactic, *flag*-ul BVI (Bistabil de Validare a Întreruperilor) este bitul 7 din registrul FLAG (figura 2.27). În terminologia de specialitate bistabilul BVI este denumit IE (*Interrupt Enable*) sau IF (*Interrupt enable Flag*) şi se implementează de regulă în variantă *read-only* în raport cu instrucţiunea LD FLAG,*src* (nu poate fi setat sau resetat prin instrucţiunea LD FLAG,*src*); poate fi setat doar prin comanda A(1)BVI („adu la 1 BVI”) activată în faza de execuţie a instrucţiunii EI şi respectiv, resetat prin comanda A(0)BVI („adu la 0 BVI”) activată în faza de execuţie a instrucţiunii DI. Tabelele de

comenzi aferente fazei de execuţie a instrucţiunilor EI şi DI sunt redate în figurile 2.66 şi respectiv 2.67.

Faza EX – instrucţiunea EI

I_1
A(1)BVI, TI$_1$ $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.66 Faza EX aferentă instrucţiunii **EI** – tabelul de comenzi

Faza EX – instrucţiunea EI

I_1
A(0)BVI, TIF, TI$_1$

Fig 2.67 Faza EX aferentă instrucţiunii **DI** – tabelul de comenzi

Să observăm că am proiectat o instrucţiune DI neinteruptibilă (tabelul de comenzi din figura 2.67 nu conţine punctul interuptibil). Din moment ce instrucţiunea DI realizează invalidarea întreruperilor este firesc să fie, ea însăşi, neinteruptibilă. Instrucţiunea EI validează întreruperile şi prin urmare, este firesc să fie, ea însăşi, interuptibilă (tabelul din figura 2.66 conţine punct interuptibil).

Reamintim faptul că instrucţiunile EI şi DI sunt instrucţiuni privilegiate (sunt „acceptate” de *hardware* şi executate numai în mod *supervisor*). În sistemele de operare protejate, validarea şi respectiv invalidarea întreruperilor sunt atribute exclusive ale sistemului de operare; aplicaţiile pot utiliza întreruperile numai prin intermediul unor funcţii de sistem (funcţii ale sistemului de operare).

Validarea/invalidarea întreruperilor cu ajutorul *flag*-ului BVI se realizează în *hardware*, aşa cum ilustrează schema de principiu din figura 2.68, unde:

INTR_G -reprezintă cererea globală de întrerupere, care se obţine printr-o funcţie SAU în care intră liniile de întrerupere aferente tuturor perifericelor din sistem. Prin urmare, dacă unul sau mai multe periferice activează linia proprie de întrerupere, va rezulta INTR_G=1.

INTR -este variabilă (semnal) de intrare în BGC. Această variabilă este testată în punctul interuptibil implementat la finele fazei de execuţie a fiecărei instrucţiuni. Dacă la testare se obţine INTR=1, se va intra în faza de întrerupere, care va suspenda temporar *task*-ul curent şi va lansa în execuţie *handler*-ul aferent perifericului care a întrerupt. INTR se poate activa în ”1” numai dacă Q_{BVI}=1 (întreruperile sunt validate); această condiţionare este realizată de poarta A din figura 2.68.

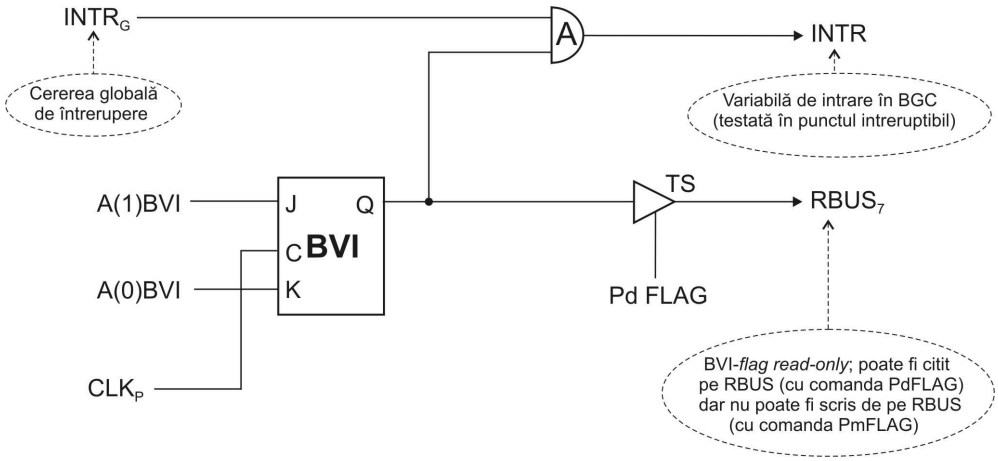


Fig 2.68 Validarea/invalidarea întreruperilor cu ajutorul *flag*-ului BVI

G. Instrucţiunile RET şi IRET

Instrucţiunea RET realizează revenirea dintr-o procedură apelată cu instrucţiunea CALL (figura 2.48). Dacă la apel (în faza EX aferentă instrucţiunii CALL), adresa de revenire (preluată practic din registrul PC) a fost salvată în stivă, la revenire (în faza EX aferentă instrucţiunii RET), această adresă de revenire trebuie restaurată din stivă în registrul PC. În concluzie, instrucţiunea RET este de fapt o instrucţiune POP PC. Tabelul de comenzi aferent instrucţiunii RET este redat în figura 2.69, fiind echivalent celui aferent instrucţiunii POP reg din figura 2.38 (doar comanda *PmRG* de pe impulsul I_2 este înlocuită cu comanda *PmPC*).

Instrucţiunea IRET realizează revenirea dintr-un *handler* de întrerupere (figura 2.9). *Handler*-ele de întrerupere sunt lansate în execuţie prin evenimente sesizate la nivel *hardware* şi semnalate (la acelaşi nivel), prin activarea unor cereri de întrerupere. Aceste cereri sunt activate de către periferice atunci când sunt constrânse să solicite anumite servicii sistemului de operare (vezi 2.3.D). În concluzie, „evenimentele” sunt detectate la nivel *hardware* iar serviciile sunt implementate la nivel *software* (*handler*-ele sunt *task*-uri din cadrul sistemului de operare).

Cu foarte puţine excepţii, instrucţiunile conţin, la finele fazei EX, aşa zisul punct interuptibil, în care se testează cererea globală de întrerupere. Dacă această cerere este activă (cel puţin un periferic solicită un anumit serviciu sistemului de operare), se va intra în faza INT, descrisă principial la 2.3.D. Faza INT va întrerupe (suspenda temporar) *task*-ul curent şi va lansa în execuţie un nou *task* (*handler*-ul de întrerupere care are sarcina de a face serviciul solicitat de periferic). În faza INT sunt salvate în stivă registrele FLAG şi PC. Prin urmere, instrucţiunea IRET, de la sfârşitul *handler*-ului va trebui să restaureze din stivă registrele PC şi FLAG (în ordine inversă, pentru a respecta principiul „*last-in-first-out*” care stă la baza stivei). Prin aceste restaurări succesive din stivă, instrucţiunea IRET realizează revenirea în *task*-ul suspendat temporar şi asigură reluarea acestuia cu starea sa

restaurată (din stivă) în registrul FLAG. În concluzie, instrucţiunea IRET este echivalentă cu o succesiune de două instrucţiuni (POP PC urmată de POP FLAG), integrate într-o singură instrucţiune. Tabelul de comenzi aferent instrucţiunii IRET este redat în figura 2.70.

Faza EX – instrucţiunea RET

I_1	I_2
PdSP, SBUS, PdALU, PmADR	*RD, PdMEM, PmPC, +2SP, TI_1* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.69 Faza EX aferentă instrucţiunii **RET** – tabelul de comenzi

Faza EX – instrucţiunea IRET

I_1	I_2	I_3	I_4
PdSP, SBUS, PdALU, PmADR	*RD, PdMEM, PmPC, +2SP*	*PdSP, SBUS, PdALU, PmADR*	*RD, PdMEM, PmFLAG, +2SP, TI_1* $\overline{\text{INTR}}$: *TIF* INTR : *TINT*

Fig 2.70 Faza EX aferentă instrucţiunii **IRET** – tabelul de comenzi

2.8 Proiectarea fazei INT

Rolul şi conţinutul fazei de întrerupere (INT) au fost descrise la 2.3.4. Descrierea de la 2.3.4. se încheie cu enumerarea operaţiilor succesive din cadrul fazei INT. Să notăm faptul că faza INT nu face parte din ciclul instrucţiunii; INT este o fază specială, parcursă numai dacă, în punctul interuptibil, de la sfârşitul instrucţiunii, se constată că există o cerere de întrerupere activă (cel puţin un periferic solicită un anumit serviciu). Serviciul solicitat va fi acordat la nivel *software*, fiind implementat într-un program special din cadrul sistemului de operare, numit *handler* de întrerupere. Iată de ce faza INT, implementată la nivel *hardware*, trebuie să suspende temporar *task*-ul curent şi să lanseze în execuţie un nou *task* (*handler*-ul de întrerupere aferent perifericului care solicită serviciu).

Reamintim faptul că aceeaşi fază INT va fi executată şi în cazul activării unei excepţii. Excepţiile, ca şi întreruperile, sunt tratate la nivel *software* (*handler*-e dedicate). La procesorul didactic am definit 2 condiţii de excepţie (ACLOW şi CIL). Detecţia acestora se face în faza IF (blocurile (3) şi respectiv (6) din figura 2.13).

2.8.1 Organigrama fazei INT

Operaţiile succesive, enumerate la finele paragrafului 2.3.4, sunt transpuse în organigrama fazei INT din figura 2.71. Explicarea organigramei INT o vom face apelând la numerotarea blocurilor din componenţa acesteia:

(1) • Se activează INTA (*INTerrupt Acknowledge*). Cu alte cuvinte, se achită întreruperea (la cererea INTR procesorul răspunde cu INTA) pentru a determina perifericul (*controller*-ul I/O aferent acestuia) să dezactiveze cererea şi respectiv SIE să genereze şi să încarce vectorul de întrerupere în registrul IVR.

• Se decrementează registrul SP cu 2 (se adaugă o nouă locaţie în vârful stivei pentru salvarea registrului FLAG).

(2) • Se transferă adresa noului vârf al stivei din SP în ADR.

(3) • Se scrie în memorie (*WR*) conţinutul registrului FLAG (STIVĂ←FLAG).

• Se decrementează registrul SP cu 2 (se adaugă o nouă locaţie în vârful stivei pentru salvarea registrului PC).

(4) • Se transferă adresa noului vârf al stivei din SP în ADR.

(5) • Se scrie în memorie (*WR*) conţinutul registrului PC (STIVĂ←PC).

(6) • Se transferă vectorul de întrerupere (VI) din IVR în ADR.

(7) • Se citeşte (*RD*) adresa de start a *handler*-ului de întrerupere din tabela IVT şi se încarcă în registrul PC (PC←MEM$|_{ADR}$).

(8),(9) • Dacă intrarea în faza INT s-a realizat din cauza activării unei excepţii (EXC=1), se resetează bistabilii de excepţie (BE), adică se resetează cauza.

(10),(11) • Dacă intrarea în faza INT s-a realizat din cauza activării unei întreruperi externe (INTR=1), se resetează bistabilii de întrerupere (BI), adică se resetează cauza.

(12) • Din faza INT se trece necondiţionat în faza IF, în cadrul căreia se va face *fetch*-ul primei instrucţiuni din *handler*.

Handler-ul este executat, instrucţiune cu înstrucţiune, ca orice program. Acesta va face, perifericului solicitant, serviciul solicitat. Exemple de servicii au fost prezentate în cadrul paragrafului 2.3.4.

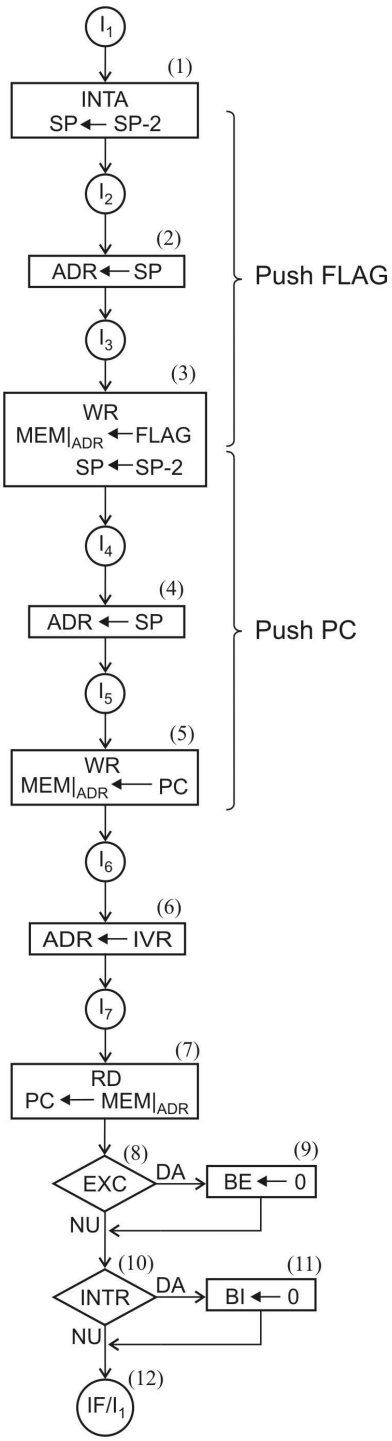


Fig 2.71 Organigrama fazei INT

2.8.2 Tabelul de comenzi aferent fazei INT

Sintetizăm operaţiile elementare din organigrama fazei INT în tabelul de comenzi aferent acestei faze (figura 2.72).

Faza INT

I_1	I_2	I_3	I_4	I_5	I_6	I_7
INTA, -2SP	*PdSP, SBUS, PdALU, PmADR*	*WR, PdFLAG, -2SP*	*PdSP, SBUS, PdALU, PmADR*	*WR, PdPC, DBUS, PdALU*	*PdIVR, PmADR*	*RD, PdMEM, PmPC, TIF, TI_1* EXC : *A(0)BE* INTR : *A(0)BI*

Fig 2.72 Tabelul de comenzi aferent fazei INT

Pentru execuţia unei anumite operaţii din organigramă trebuie activate una sau mai multe comenzi elementare (C_L). Coloanele tabelului de comenzi corespund celor şapte impulsuri de stare (perioade de tact) din componenţa fazei INT. Fiecare coloană din tabel va conţine comenzile logice elementare ce se vor activa pe impulsul de stare respectiv. Aceste comenzi se deduc din organigramă. Pentru fiecare operaţie elementară din organigramă, trebuie dedus şi trecut în tabel setul de comenzi elementare aferent. Pentru transpunerea corectă a organigramei în tabel de comenzi trebuie utilizată schema bloc a procesorului didactic (figura 2.2). Pe această schemă bloc, se deduce cu uşurinţă **setul de comenzi elementare** aferent unei anumite **operaţii elementare**.

Faza INT se execută în 7 impulsuri de stare ($I_1 \div I_7$). Fiecare impuls de stare marchează o perioadă de tact şi prin urmare, cele 7 impulsuri de stare succesive marchează cele 7 perioade de tact din componenţa fazei INT.

Comenzile aferente unui anumit impuls de stare (comenzile înscrise pe o anumită coloană a tabelului de comenzi) sunt **comenzi concurente !** Impulsul de stare marchează o perioadă de tact şi tote comenzile asignate (în tabel) acelui impuls de stare se activează pe durata respectivei perioade de tact.

În general comenzile sunt de tip nivel; excepţie fac comenzile „*primeşte registru*" care sunt de tip impuls (fig 2.18). Forma comenzilor (de tip nivel şi respectiv impuls) precum şi activarea concurentă a comenzilor asignate unui anumit impuls de stare, permit efectuarea corectă a operaţiilor elementare. Reamintim faptul că toate operaţiile registru-registru (cu eventuala procesare în ALU a informaţiilor transferate) sunt operaţii elementare (operaţii efectuate pe durata unei singure perioade de tact). Explicaţii detaliate relative la efectuarea operaţiilor elementare urmeză a fi redate în cadrul paragrafului dedicat implementărilor *hardware*, în mod special în cadrul paragrafului 2.9.4 (fig 2.81 şi 2.82).

Organigramele şi tabelele de comenzi realizează o descriere amănunţită (la nivel de perioadă de tact) a fazelor. Primele 3 faze (IF, OF şi EX) formează ciclul instrucţiunii. Cea de-a patra fază (INT) este o fază specială, parcursă la activarea unei întreruperi sau excepţii (fig 2.12). Decizia de a recurge la tabele de comenzi a avut o dublă motivaţie:

1. intenţia de a pune la dispoziţia cititorului un instrument care să permită o descriere amănunţită a modului de procesare a fiecărei instrucţiuni.

2. necesitatea de a sintetiza într-o manieră eficientă şi expresivă, toate informaţiile care se vor dovedi necesare pentru proiectarea unităţii de control a procesorului. Este important de subliniat că proiectarea unităţii de control cablate (BGC-ului cablat) se va face pornind de la tabelele de comenzi elaborate.

2.9 Implementarea în *hardware* a elementelor procesorului didactic

În paragraful 2.2 a fost prezentată schema bloc a procesorului didactic (figura 2.2), în cadrul căreia au fost identificate următoarele elemente componente:

- busurile interne (SBUS, DBUS, RBUS) precum şi busul memoriei (MBUS)
- unitatea aritmetico-logică (ALU)
- registrele interne:
  - registrele generale (R_{15}÷R_0)
  - registrele speciale (IR, SP, A, M, PC, FLAG, ADR, IVR)
- unitatea de control (BGC), compusă din:
  - generatorul de faze (GF)
  - generatorul impulsurilor de stare (GIS)
  - rețeaua logică combinațională (RLC)
- sistemul de întreruperi şi excepţii (SIE)

În acest paragraf vom descrie pe scurt implementările *hardware* aferente acestor blocuri componente.

2.9.1 Tipuri de comenzi

În paragraful 2.2 au fost identificate şi descrise pe scurt comenzile elementare (C_L) cu ajutorul cărora se controlează operaţiile elementare executate în cadrul procesorului. În principiu, operaţia elementară este operaţia care se execută într-o singură perioadă de tact. În organigramele proiectate pe parcursul paragrafelor 2.5÷2.8 pot fi identificate operaţii elementare iar în tabelele de comenzi corespondente pot fi identificate comenzile (C_L) care conduc la efectuarea acestor operaţii.

Din punct de vedere al duratei (timpului în care comanda este menţinută activă), comenzile elementare se clasifică în două categorii:

- comenzi de tip nivel (active pe toată durata perioadei de tact)
- comenzi de tip impuls (active pe durata impulsului de tact)

În figura 2.73 este redată forma acestor comenzi.

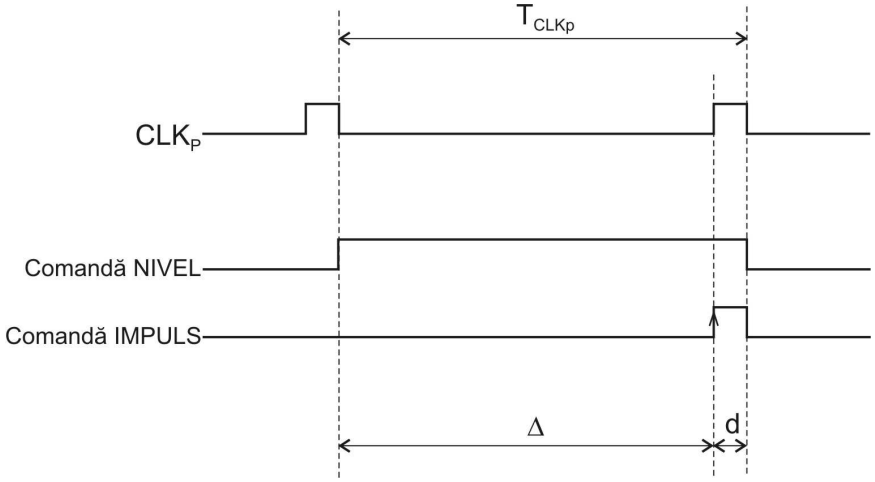


Figura 2.73 Comenzi de tip nivel şi comenzi de tip impuls

Comenzile „*Primeşte Registru*” sunt comenzi de tip impuls. Încărcarea datelor în registrul specificat (*PmA*, *PmIR*, *PmPC*, *PmADR*, etc) se face pe front (frontul urcător aferent semnalului de comandă).

Celelalte comenzi, cum ar fi: comenzile „*Predă Registru*” (*PdA*, *PdIR[OP]*, *PdM*, etc), comenzile destinate unităţii ALU (*SUM*, *SBUS*, *AND*, *XOR*, C_{in}, etc), comanda *PdALU*, comenzile de citire şi respectiv scriere din/în memorie (*RD* şi *WR*), sunt comenzi de tip nivel.

După cum rezultă din figura 2.73, comenzile de tip nivel se activează chiar la începutul perioadei de tact, în timp ce comenzile de tip impuls se activează o dată cu frontul urcător al impulsului de tact; între momentul activării comenzii nivel şi respectiv cel al activării comenzii impuls apare o întârziere semnificativă, notată cu Δ în figura 2.73. O operaţie de transfer **Registru sursă→Registru destinaţie** se realizează cu implicarea busurilor interne si a unitătii ALU. Cu ajutorul comenzii „*Predă Registru*” (de tip nivel) se emit datele din registrul sursă, tot cu ajutorul unor comenzi de tip nivel se asigură traversarea unităţii ALU şi respectiv emisia datelor pe RBUS, de unde cu comanda „*Primeşte Registru*” (de tip impuls), datele transferate se încarcă în registrul destinaţie (vezi figura 2.2). Suma întârzierilor cumulate care afectează datele transferate (pe traseul parcurs de la emisia datelor din registrul sursă şi până la încărcarea acestora în registrul destinaţie), trebuie să fie ≤ Δ. Dacă această condiţie este îndeplinită, în registrul destinaţie vor fi încărcate date corecte. Dacă condiţia este neîndeplinită, în registrul destinaţie vor fi încărcate date incorecte (deoarece cei 16 biţi transferaţi ajung pe intrarea registrului destinaţie după frontul urcător al comenzii „*Primeşte Registru*”). O astfel de analiză se va face pentru determinarea perioadei minime (frecvenţei maxime) aferente semnalului de tact CLK_P (paragraful 2.9.4).

2.9.2 Conectarea registrelor la busuri

Procesorul didactic are 3 busuri interne pe 16 biţi (SBUS, DBUS şi RBUS). Un bus pe 16 biţi este format din 16 linii de semnal cărora li se ataşează circuitele de acces la bus. Aceste circuite de acces conectează registrele la busul respectiv. De exemplu, la DBUS sunt conectate registrele speciale M şi respectiv PC şi registrele generale $R_{15} \div R_0$. Circuitele de acces la DBUS trebuie să asigure conectarea tuturor acestor surse (registre) la cele 16 linii aferente DBUS-ului. Prin urmare, circuitele de acces trebuie să realizeze o funcţie logică SAU între toate aceste surse. În figura 2.74 este explicitată conectarea registrelor M şi respectiv PC la DBUS.

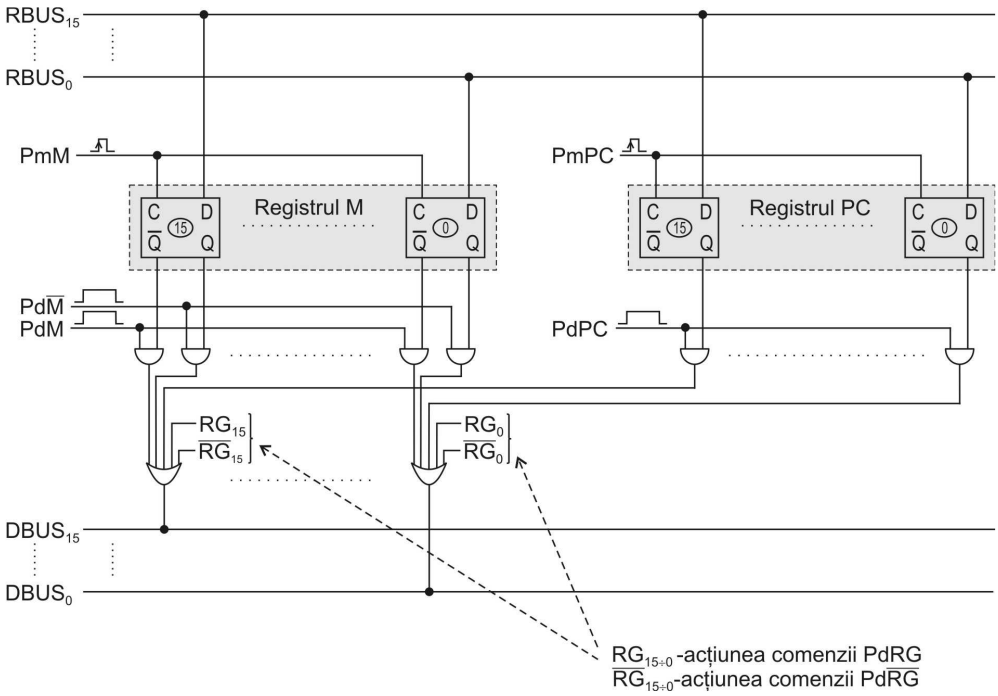


Fig 2.74 Conectarea registrelor M şi PC la DBUS

După cum rezultă din schema bloc aferentă procesorului didactic (figura 2.2), registrele M şi respectiv PC pot fi încărcate cu date preluate de pe RBUS şi pot fi citite (emise) pe DBUS. Aşa cum rezultă din figura 2.74, pentru încărcare (scriere în registru) se utilizează comenzile de tip impuls *PmM* şi respectiv *PmPC*. Comanda *PmM* (*PmPC*) se aplică pe intrarea C (de *clock*) aferentă celor 16 bistabili ce compun registrul M (PC) iar cei 16 biţi de date preluaţi de pe RBUS se aplică pe intrarea D aferentă celor 16 bistabili. Încărcarea datelor în registru (scrierea în registru) se va realiza pe frontul urcător al comenzii *PmM* (*PmPC*). Se realizează astfel o operaţie sincronă de încărcare paralelă.

Citirea registrului M (PC) se realizează cu comanda de tip nivel *PdM* (*PdPC*). Prin citire, cei 16 biţi de date stocaţi în registru, sunt emişi pe DBUS. Comanda *PdM* (*PdPC*) validează cele 16 porţi SI conectate la ieşirile Q aferente celor 16 bistabili ce compun registrul M (PC). Prin urmare, cei 16 biţi de date memoraţi în registru traversează cele 16 porţi SI, apoi cele 16 porţi SAU, ajungând astfel pe cele 16 linii aferente DBUS-ului. În concluzie, conţinutul registrului M (PC) va fi prezent pe DBUS atâta timp cât comanda *PdM* (*PdPC*) este activă (în "1"). De aici rezultă că, un singur registru poate fi emis pe DBUS la un moment dat (pe durata unei anumite perioade de tact). Prin urmare, nu pot fi activate simultan (pe aceeaşi perioadă de tact) ambele comenzi, *PdM* şi respectiv *PdPC*, deoarece, prin activarea simultană a celor două comenzi, ar rezulta conflict pe DBUS. Această regulă trebuie respectată la nivelul fiecărui bus şi respectiv fiecărei perioade de tact. Cu alte cuvinte, comenzile *„Predă Registru"* aferente registrelor conectate la un anumit bus, se exclud reciproc (o singură astfel de comandă poate fi activată pe durata unei perioade de tact).

Comanda $Pd\overline{M}$ emite pe DBUS, conţinutul registrului M, inversat bit cu bit (complementul faţă de unu al numărului din M). Implementarea comenzii $Pd\overline{M}$ din figura 2.74 este în acord cu cerinţele impuse de schema bloc a procesorului (figura 2.2). Comanda $Pd\overline{M}$ este utilizată în faza de execuţie aferentă instrucţiunii SUB dest,src (paragraful 2.7.1.C).

În schema bloc din figura 2.2 se observă că şi setul de registre generale (R_{15}÷R_0) este conectat la DBUS. În paragraful 1.4.1 (figura 1.8) este descris modul de implementare a unui set de registre generale într-o structură uniport. O structură uniport de memorie statică rapidă, cu 16 locaţii (16 registre) pe 16 biţi revendică şi setul de registre generale din structura procesorului didactic. Pentru structura uniport cu 16 locaţii (16 registre) este necesar un singur set de 4 linii de adresă. Adresa pe 4 biţi, aplicată pe cele 4 linii de adresă, va selecta un registru care poate fi citit, cu o comandă de tip nivel (PdRG) şi respectiv scris, cu o comandă de tip impuls (PmRG).

Analizând schema bloc a procesrului didactic (figura 2.2), constatăm că există două comenzi de citire care operează la nivelul setului de registre generale: comanda *PdRG* emite pe DBUS conţinutul nemodificat al registrului (selectat în cadrul setului) iar comanda $Pd\overline{RG}$ emite pe DBUS conţinutul negat (inversat bit cu bit). Din acest punct de vedere există deci o similitudine perfectă cu registrul M. Prin urmare, circuitele de conectare la DBUS aferente setului de registre generale vor fi identice cu cele aferente registrului M.

Selecţia registrului (citit pe DBUS sau scris de pe RBUS) se face la nivel *hardware*, cu ajutorul celor 4 biţi de adresă codificaţi în codul instrucţiunii curente (câmpul ADR. REG. din figurile 2.5. b), 2.5.c) şi respectiv 2.5.d). Cei patru biţi ai câmpului ADR. REG. se aplică pe cele patru intrări ale decodificatorului de adrese aferent setului de registre generale. Prin urmare, registrul specificat de instrucţiune va fi selectat automat (în cadrul setului de registre) atâta timp cât codul instrucţiunii respective este prezent în IR. Registrul general specificat de o instrucţiune, va fi utilizat pe parcursul fazei *fetch* operand (uneori şi pe parcursul fazei de execuţie). Selecţia registrului este evident asigurată, pe parcursul ambelor faze, deoarece codul instrucţiunii curente va fi evacuat din IR doar după încheierea fazei de execuţie (prin suprascriere cu codul următoarei instrucţiuni, o dată cu *fetch*-ul acesteia).

Setul de registre generale nu este prezentat în figura 2.74; sunt figurate (pe intrările porţilor SAU) doar ieşirile de date $RG_{15\div0}$ şi respeciv $\overline{RG}_{15\div0}$. Implementarea în

hardware a comenzilor *PdRG* şi respectiv $Pd\overline{RG}$ se va realiza întro manieră perfect similară celei în care operează comenzile *PdM* şi respectiv $Pd\overline{M}$.

Schema de conectare a registrelor la DBUS din figura 2.74 poate fi reproiectată utilizând circuite cu trei stări (ieşiri *tree-state*). O astfel de soluţie este prezentată în figura 2.75.

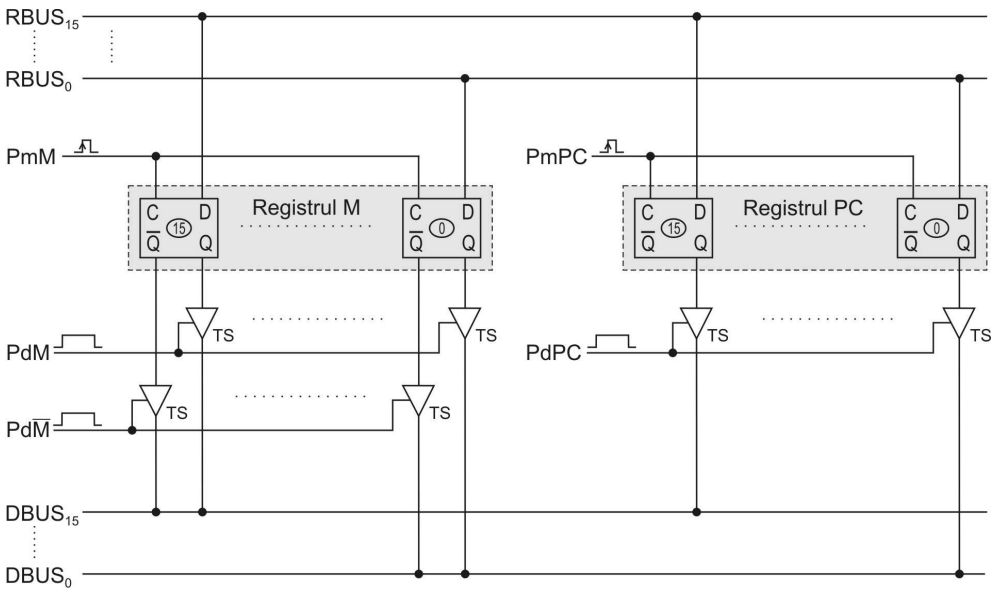


Fig 2.75 Conectarea registrelor M şi PC la DBUS utilizând circuite cu ieşiri *three-state*

Cele două scheme (din figurile 2.75 şi respectiv 2.74) sunt echivalente. Circuitele cu ieşiri *three-state* realizează o funcţie SAU-CABLAT pe DBUS. Prin urmare, nu pot fi activate simultan (pe aceeaşi perioadă de tact) două sau mai multe comenzi „*Predă Registru*”, deoarece ar rezulta conflict pe DBUS. În cazul schemei din figura 2.75, dacă pe o anumită perioadă de tact, nici-o comandă „*Predă Registru*” nu este activată, atunci liniile DBUS-ului vor fi în stare de înaltă impedanţă (în raport cu toate circuitele *three-state* conectate la DBUS).

2.9.3 Unitatea ALU

Procesorul didactic are o structură orientată pe acumulator. Anumite operaţii, cum sunt operaţiile de deplasare şi rotire (vezi faza de execuţie aferentă instrucţiunilor de deplasare şi rotire descrisă la 2.7.2.D) au fost implementate direct pe acumulator (pe registrul A). Prin urmare, în unitatea ALU vor trebui implementate doar operaţiile neimplementate pe registrul A. Făcând un inventar al tuturor tabelelor de comenzi elaborate (paragrafele 2.5÷2.8), obţinem setul de comenzi şi respectiv operaţii pe care trebuie să le implementăm în unitatea ALU. Acestea sunt prezentate în tabelul 2.7.

Comanda	Operaţia ALU	
SUM	**Adunare**	(ALU generează suma operanzilor preluaţi de pe SBUS şi respectiv DBUS)
AND	**SI logic**	(ALU realizează operaţia SI pe 16 biţi, între cei doi operanzi preluaţi de pe SBUS şi respectiv DBUS)
OR	**SAU logic**	(ALU realizează operaţia SAU pe 16 biţi, între cei doi operanzi preluaţi de pe SBUS şi respectiv DBUS)
XOR	**SAU exclusiv**	(ALU realizează operaţia SAU EXCLUSIV pe 16 biţi, între cei doi operanzi preluaţi de pe SBUS şi respectiv DBUS)
SBUS	**Transfer sursă**	(ALU transferă la ieşiri, fără nici-o modificare, operandul preluat de pe SBUS)
$\overline{SBUS}$	**Negare sursă**	(ALU transferă la ieşiri, negat bit cu bit, operandul preluat de pe SBUS)
DBUS	**Transfer destinaţie**	(ALU transferă la ieşiri, fără nici-o modificare, operandul preluat de pe DBUS)

Tabelul 2.7 Comenzile ALU şi operaţiile aferente

Să notăm că ALU este un bloc combinaţional. ALU nu conţine bistabili (elemente de memorare); ALU conţine doar logică combinaţională (porţi logice). Pe baza setului de comenzi din tabelul 2.7, rezultă schema bloc aferentă unităţii ALU (figura 2.76)

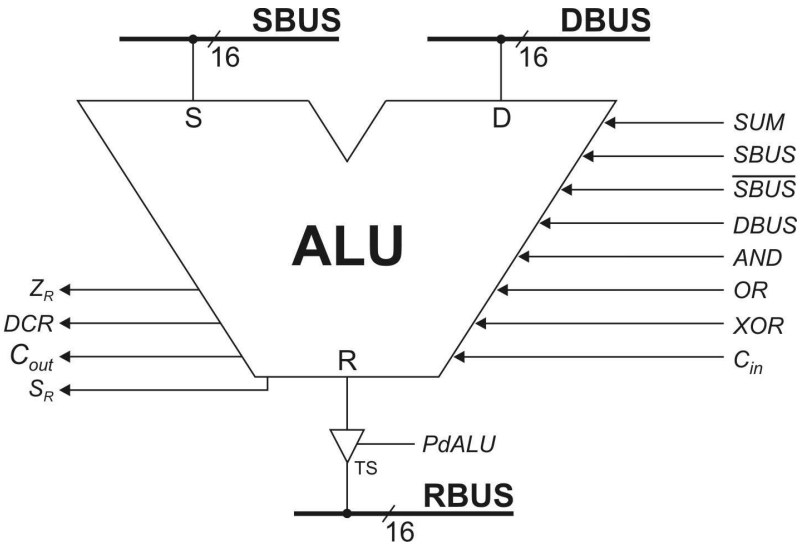


Fig 2.76 Schema bloc aferentă unităţii ALU

Aşa cum indică figura 2.76, ALU preia cei doi operanzi pe 16 biţi de pe SBUS şi respectiv DBUS (intrările S şi respectiv D), execută operaţia specificată de comanda activată, generează rezultatul pe 16 biţi pe ieşirea R, de unde prin activarea comenzii *PdALU*, acesta va fi emis pe RBUS. Suplimentar, ALU generează cele patru semnale de condiţii (C_{out}, Z_R, S_R, *DCR*) care, la instrucţiunile aritmetice, vor poziţiona cele patru *flag*-uri de condiţii din registrul FLAG (bistabilii C, Z, S, V). Poziţionarea *flag*-urilor a fost prezentată la 2.7.1.B (vezi figura 2.28 şi explicaţiile aferente).

Unitatea ALU trebuie să execute 7 operaţii (tabelul 2.7). Fiecare opeaţie va fi efectuată într-un bloc dedicat. Prin urmare, ALU va conţine 7 blocuri (figura 2.77).

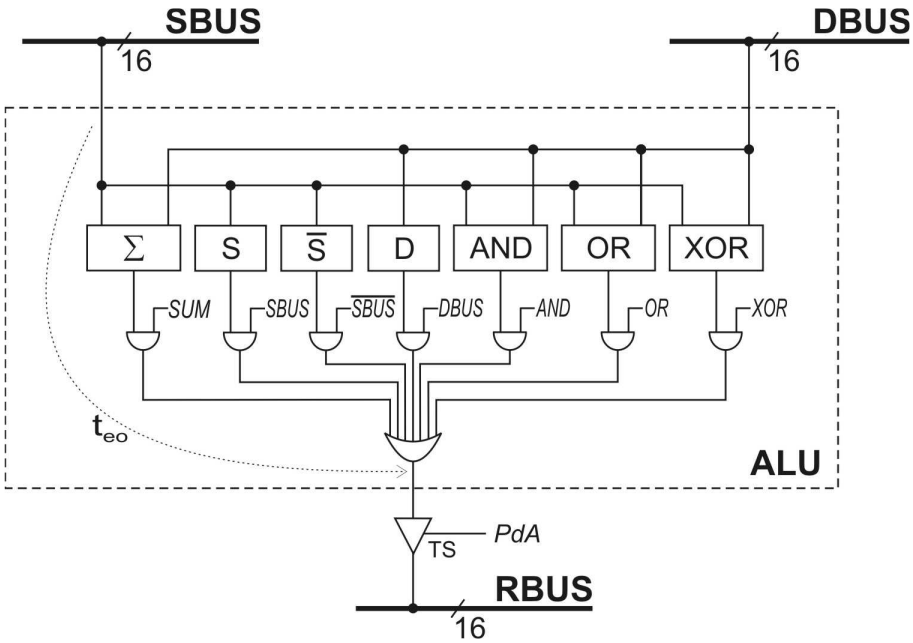


Fig 2.77 Structura unităţii ALU

Blocurile Σ (sumator), AND, OR şi XOR efectuează operaţii cu doi operanzi. Aceste blocuri au prin urmare două intrări. Blocurile S, $\overline{S}$ şi D efectuează operaţii asupra unui singur operand. Aceste blocuri au prin urmare câte o singură intrare. Rezultatul obţinut pe ieşirea unui bloc va fi emis (validat) pe ieşirea ALU cu ajutorul comenzii corespondente.

În figura 2.77 am notat cu t_{eo} timpul de efectuare a unei operaţii în ALU. Acesta este timpul de propagare a datelor (semnalelor logice) prin *hardware*-ul din cadrul unitatăţii ALU (timpul scurs din momentul aplicării operanzilor pe intrările ALU şi până la obţinerea rezultatului pe ieşirea acestuia). Cele 7 comenzi (*SUM*, *SBUS*, $\overline{SBUS}$, *DBUS*, *AND*, *OR* şi *XOR*) sunt comenzi de tip nivel (figura 2.73). La activare, aceste comenzi sunt active pe toată durata perioadei de tact. Pe de altă parte, operanzii (aplicaţi pe intrările ALU) sunt emişi pe SBUS şi respectiv DBUS prin activarea unor comenzi de tip „*Predă Registru*”. Acestea sunt tot comenzi de tip nivel. Prin urmare, atât comenzile ALU cât şi comenzile din amonte sunt comenzi de tip nivel. Aşa cum indică figura 2.73, comenzile de tip nivel se

activează chiar la începutul perioadei de tact. Asta înseamnă că până la obţinerea rezultatului pe ieşirea ALU, vor fi traversate două circuite (întârzieri) succesive:

- circuitele de acces pe SBUS/DBUS (timpul de întârziere pe SBUS/DBUS)
- circuitele aferente blocului activat în cadrul unităţii ALU (timpul de efectuare a operaţiei în ALU, pe care l-am notat cu t_{eo} în figura 2.77).

Semnalele C_{in} şi respectiv C_{out} sunt legate de sumatorul (Σ) din cadrul unităţii ALU; C_{in} reprezintă transportul de intrare aferent acestui sumator pe 16 biţi (transport de intrare în rangul cel mai puţin semnificativ) iar C_{out} reprezintă transportul de ieşire din acest sumator (transportul de ieşire din rangul cel mai semnificativ). Semnalul C_{out} (semnal de ieşire) este utilizat la poziţionarea *flag*-ului C. Să notăm că C_{in} (semnal de intrare) reprezintă de fapt o comandă pentru ALU. Această comandă poate fi utilizată, împreună cu comanda *SUM*, pentru operaţii de tipul A+B+1 / A+1 / B+1; vezi faza de execuţie aferentă instrucţiunilor INC src (paragraful 2.7.1.F) şi respectiv INC A (paragraful 2.7.2.A).

Blocul Σ din componenţa ALU este un sumator pe 16 biţi. Operaţia de adunarea revendică propagarea transportului şi prin urmare sumatorul obişnuit (cu propagarea serială a biţilor de transport, din rang în rang) este un sumator lent, conducând la un timp de efectuare a operaţiei (t_{eo}) inacceptabil de mare. Din acest motiv, proiectanţii recurg la implementări mult mai rapide (dar şi semnificativ mai complexe), cum ar fi de exemplu sumatorul cu generare anticipată (directă) a biţilor de *carry* (*carry lookahead adder*). Structura unui astfel de sumator (pe 4 biţi) este redată în figura 2.78.

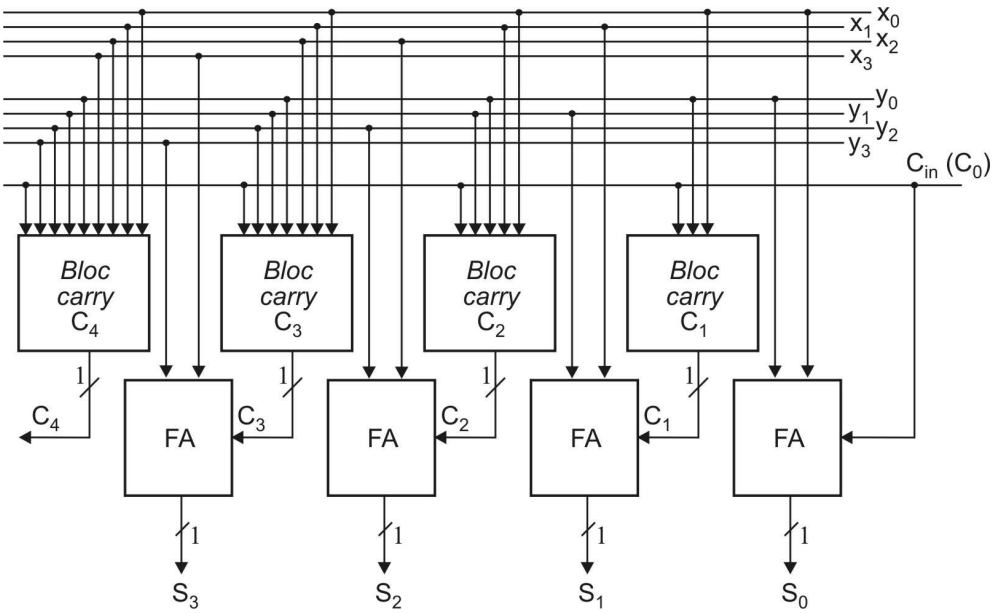


Fig 2.78 Structura sumatorului cu generare directă a biţilor de transport (*carry lookahead adder*)

Sumatorul din figura 2.78 adună doi operanzi pe 4 biţi ($x_3x_2x_1x_0$ şi respectiv $y_3y_2y_1y_0$), cu transportul de intrare ($C_{in}=C_0$) şi generează suma $S_3S_2S_1S_0$ (pe patru biţi) şi transportul de ieşire $C_{out}=C_4$. Un sumator pe 16 biţi, aşa cum necesită procesorul didactic, ar avea o structură echivalentă cu cea din figura 2.78, dar ar avea lungimea de 4 ori mai mare deoarece trebuie să adune operanzi pe 16 biţi.

Modulele FA din figura 2.78 sunt sumatoare pe un bit (*Full Adder*). Structura modulului FA, împreună cu ecuaţiile care definesc ieşirile S_i şi respectiv C_{i+1} este redată în figura 2.79.

Cele 4 blocuri de *carry* din figura 2.78 generează direct biţii de *carry* (C_{in}) pentru cele 4 sumatoare FA. Să notăm că intrările în blocurile de *carry* sunt reprezentate doar de biţii celor doi operanzi şi de transportul de intrare C_0. Funcţia fiecărui bloc de *carry* se obţine pornind de la expresia ieşirii $C_{out}=C_{i+1}$ din figura 2.79.

$$C_{i+1}=x_iy_i+C_i(x_i+y_i) \qquad (2.7)$$

Deci, C_1 poate fi generat conform expresiei:

$$C_1=x_0y_0+C_0(x_0+y_0)$$

În mod similar, C_2 va fi:

$$C_2=x_1y_1+C_1(x_1+y_1)=x_1y_1+[x_0y_0+C_0(x_0+y_0)](x_1+y_1)$$

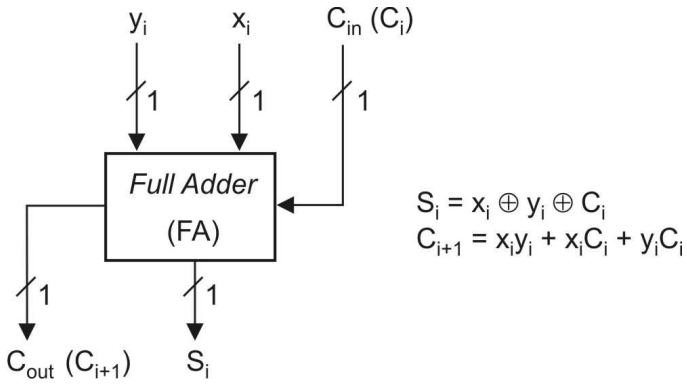


Fig 2.79 Structura sumatorului pe un bit (*Full Adder*)

Pentru simplificarea expresiilor C_i, se utilizează adesea notaţiile:

$$g_i=x_iy_i$$
$$p_i=x_i+y_i$$

În aceste condiţii ecuaţia (2.7) devine:

$$C_{i+1}=g_i+p_iC_i \qquad (2.8)$$

unde:

g_i -<u>transport generat</u>. Este transportul generat chiar în rangul considerat (rangul i). Prin urmare, transportul de ieşire din rangul i (C_{i+1}) va fi 1 atunci când transportul generat în rangul i (**g_i**) este 1

p_i -<u>transport propagat</u>. Reprezintă de fapt condiţie de propagare (peste rangul i) a transportului de inrare C_i. Prin urmare, dacă transportul de intrare este 1 (C_i=1) şi dacă există şi condiţie de propagare peste rangul i (p_i=1), atunci se va obţine C_{i+1}=1. Cu alte cuvinte, C_{i+1} va fi 1 atunci când p_i şi C_i sunt simultan 1.

Utilizând aceste notaţii, obţinem:

$$\begin{aligned}
&C_1=g_0+C_0p_0\\
&C_2=g_1+p_1g_0+p_1p_0C_0 \qquad (2.9)\\
&C_3=g_2+p_2g_1+p_2p_1g_0+p_2p_1p_0C_0\\
&C_4=g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0+p_3p_2p_1p_0C_0\\
&\quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\end{aligned}$$

Ecuaţiile (2.9) vor fi implementate în cele 4 blocuri de *carry* din figura 2.78.

Este evident faptul că structura blocurilor de *carry* se complică pe măsură ce ne deplasăm spre ranguri superioare. În cazul procesorului didactic (sumator pe 16 biţi), ultimul bloc de *carry* va genera $C_{out}=C_{15}$. Acest bloc de *carry* va avea pe intrări:

- $x_{15\div0}$ şi respectiv $y_{15\div0}$; reprezintă biţii din componenţa celor doi operanzi (în total 2×16 biţi).

- $C_{in}=C_0$; reprezintă transportul de intrare în rangul cel mai puţin semnificativ.

Este evident faptul că, în cadrul sistemului de ecuaţii (2.9), ecuaţia ieşirii $C_{out}=C_{15}$ va fi cea mai complexă.

Structura sumatorului cu generare directă a biţilor de *carry* este complexă dar, pe de altă parte, induce un timp redus de efectuare a operaţiei de adunare în ALU (t_{eo} mic). Acest timp va fi luat în considerare la determinarea perioadei minime aferentă semnalului de tact. Inversul perioadei este frecvenţa. Prin urmare, timpul t_{eo} este unul din parametrii care determină frecvenţa maximă (pe intrarea de *clock*) acceptată de procesor.

Celelalte şase blocuri (S, $\overline{S}$, D, AND, OR şi XOR) din componenţa ALU efectuează operaţii mai simple şi prin urmare, au o structură mult mai simplă (figura 2.80).

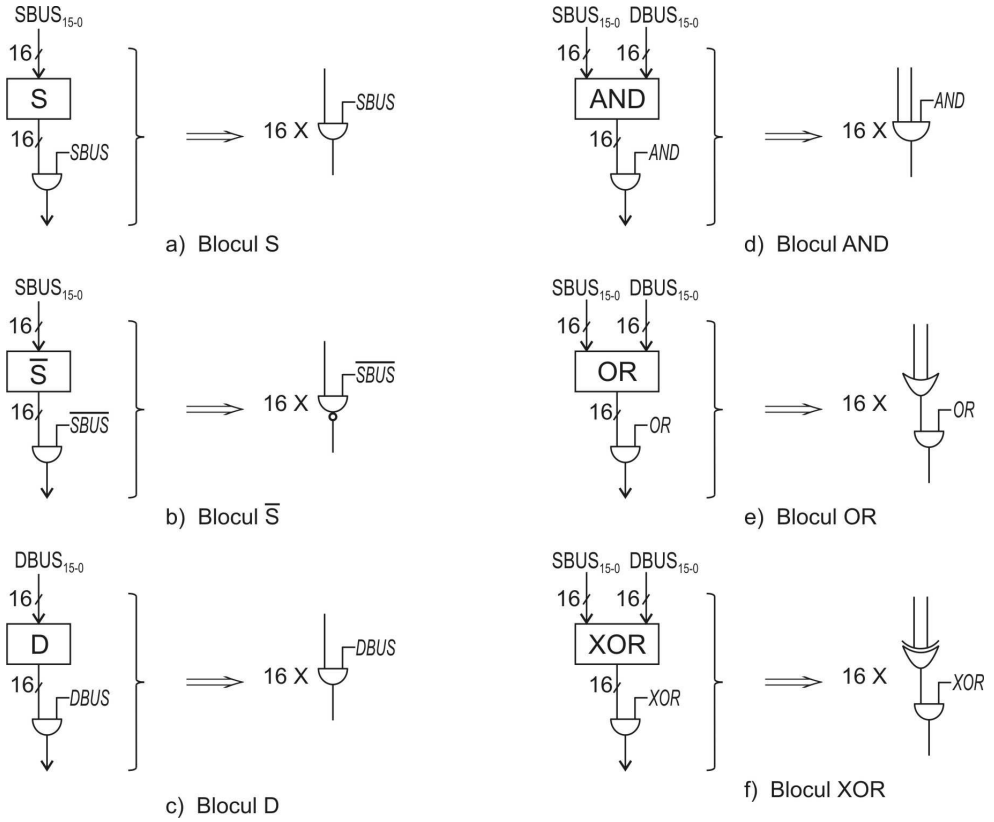


Fig 2.80 Structura blocurilor S, $\overline{S}$, D, AND, OR şi XOR din componenţa ALU

Blocurile S şi D au structuri identice, fiind formate dintr-un set de 16 porţi SI cu două intrări. Blocul $\overline{S}$ este format dintr-un set de 16 porţi SI-NU cu două intrări. Blocul AND, într-o variantă optimizată, este format dintr-un set de 16 porţi SI cu trei intrări. Blocul OR este format dintr-un set de 16 porţi SAU (cu două intrări) şi un set de 16 porţi SI (cu două intrări) iar blocul XOR este format dintr-un set de 16 porţi SAU-EXCLUSIV (cu două intrări) şi un set de 16 porţi SI (cu două intrări).

Observaţie

La procesoarele orientate pe registre generale (fără acumulator), operaţiile de deplasare şi respectiv rotire se implementează tot în ALU. Prin urmare, în structura unităţii ALU va apărea un bloc combinaţional special dedicat operaţiilor de deplasare şi rotire (blocul SHIFT). Structura de principiu a blocului SHIFT a fost prezentată la 2.7.2.D (vezi observaţia 2).

2.9.4 Determinarea frecvenţei semnalului de tact

Performanţa unui procesor (viteza de procesare) este direct proporţională cu frecvenţa semnalului de tact (f_{CLK}). Viteza creşte o dată cu creşterea frecvenţei (reducerea perioadei T_{CLK}). Limita inferioară aferentă parametrului T_{CLK} este determinată de anumiţi timpi de întârziere (propagare) din interiorul procesorului (figura 2.81).

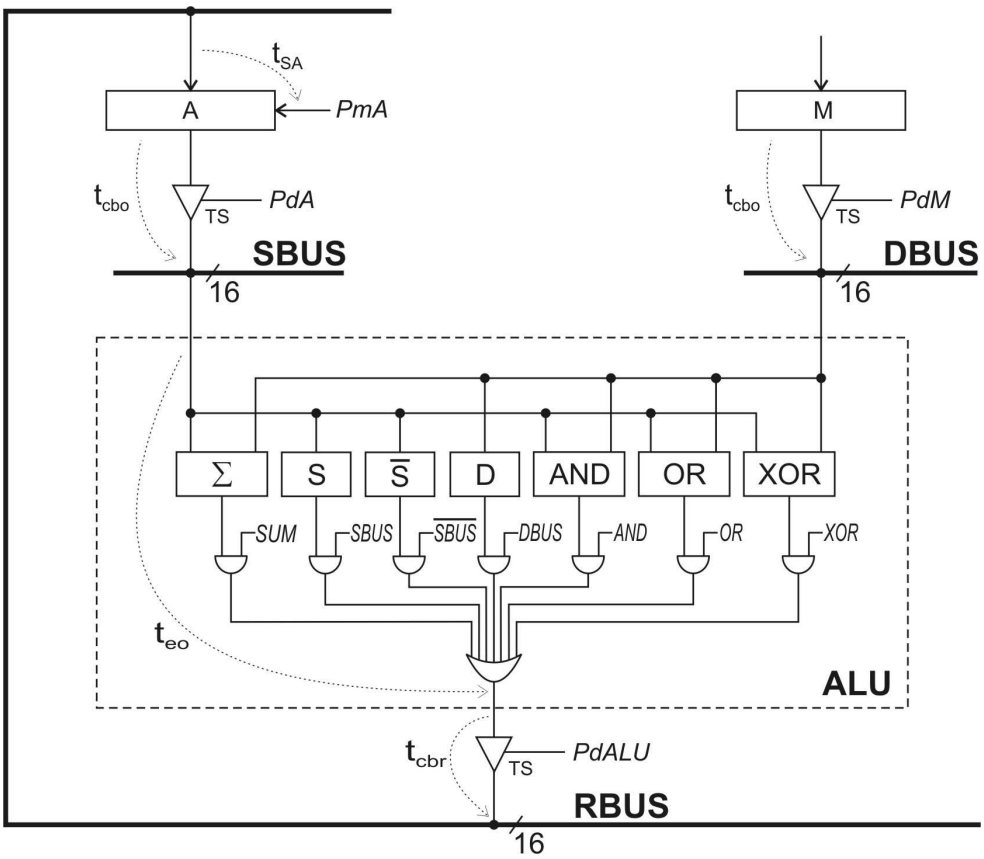


Fig 2. 81 Structurile *hardware* implicate în execuţia operaţiei A+M→A

Pentru determinarea parametrului T_{CLKmin} trebuie luată în considerare operaţia ALU care consumă cel mai mult timp. Având în vedere complexitatea ridicată a blocului Σ din componenţa ALU, este de aşteptat ca operaţia cea mai costisitoare (din punctul de vedere al timpului de execuţie) să fie adunarea.

Vom nota cu t_{eo} timpul de execuţie a operaţiei de adunare în ALU. Să considerăm operaţia elementară A+M→A executată în procesorul didactic (figura 2.81). La efectuarea acestei operaţii de adunare, pe lângă timpul t_{eo} mai intervin următorii timpi (figura 2.81):

- t_{gc} -timp de generare comandă. Unitatea de control (BGC) induce ea însăşi un timp de întârziere în generarea comenzilor. Pentru execuţia operaţiei de adunare (A+M→A), unitatea de control activează patru comenzi de tip nivel (*PdA*, *PdM*, *SUM* şi *PdALU*) şi o comandă de tip impuls (*PmA*). Dacă comenzile *PdA* şi respectiv *PdM* se generează cu o întârziere de t_{gc} nanosecunde (în raport cu începutul perioadei de tact considerate), atunci validarea circuitelor de acces la SBUS/DBUS se iniţiază exact cu aceeaşi întârziere. (figura 2.82).

- t_{cbo} -timp de conectare la bus a operanzilor (întârzierea pe care o introduc circuitele de acces la SBUS/DBUS).

- t_{cbr} -timp de conectare la bus a rezultatului (întârzierea pe care o introduc circuitele de acces la RBUS).

- t_{SA} -timp de stabilizare a informaţiei pe intrările de date aferente registrului destinaţie (registrului A). Pentru a avea certitudinea încărcării corecte a rezultatului în registrul A, cei 16 biţi ai rezultatului trebuie aplicaţi pe intrarea D aferentă celor 16 bistabili ce compun registrul A, cu t_{SA} nanosecunde înaintea frontului activ al comenzii *PmA*; comanda *PmA* se aplică pe intrarea C aferentă celor 16 bistabili.

- d -durata impulsului de tact.

În figura 2.82 este reprezentată perioada de tact pe care se efectuează operaţia de adunare A+M→A.

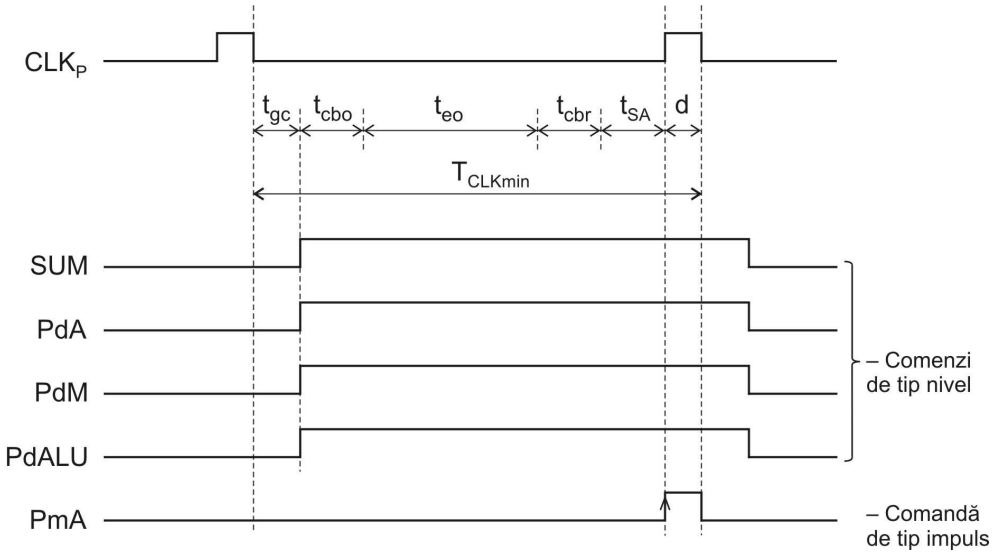


Fig 2.82 Întârzierile implicate în execuţia operaţiei A+M→A

În cadrul perioadei de tact considerate, pot fi identificaţi toţi timpii de întârziere enumeraţi mai sus (t_{gc}, t_{cbo}, t_{eo}, t_{cbr} şi respectiv t_{SA}). Pe baza figurii 2.82 se poate determina perioada minimă aferentă semnalului de tact (ecuaţia 2.10).

$$T_{CLKmin}=t_{gc}+t_{cbo}+t_{eo}+t_{cbr}+t_{SA}+d \qquad (2.10)$$

Frecvenţa maximă a tactului procesorului va fi inversul perioadei minime: $f_{max}=1/T_{CLKmin}$
Prin urmare, pentru a creşte performanţa procesorului trebuie acţionat în sensul reducerii valorilor timpilor de întârziere (factorul tehnologic).

A două direcţie pe care se acţionează este reprezentată de introducerea paralelismului arhitectural (*pipeline*-ul de la nivelul instrucţiunii şi de la nivelul operaţiilor aritmetice). Paralelismul arhitectural (procesarea în tehnică *pipeline*) este prezentă în toate procesoarele moderne. Principial, procesoarele superscalare conţin chiar mai multe cascade *pipeline* care operează concurent; sunt prin urmare arhitecturi capabile să lanseze în execuţie, simultan, mai multe instrucţiuni.

2.10 Proiectarea unităţii de control

Unitatea de control aferentă procesorului generează comenzile elementare (C_L) prin care se declanşează şi se execută operaţiile elementare (operaţiile executabile pe durata unei singure perioade de tact). Din acest motiv unitatea de control se mai numeşte şi **bloc de generare comenzi** (BGC).

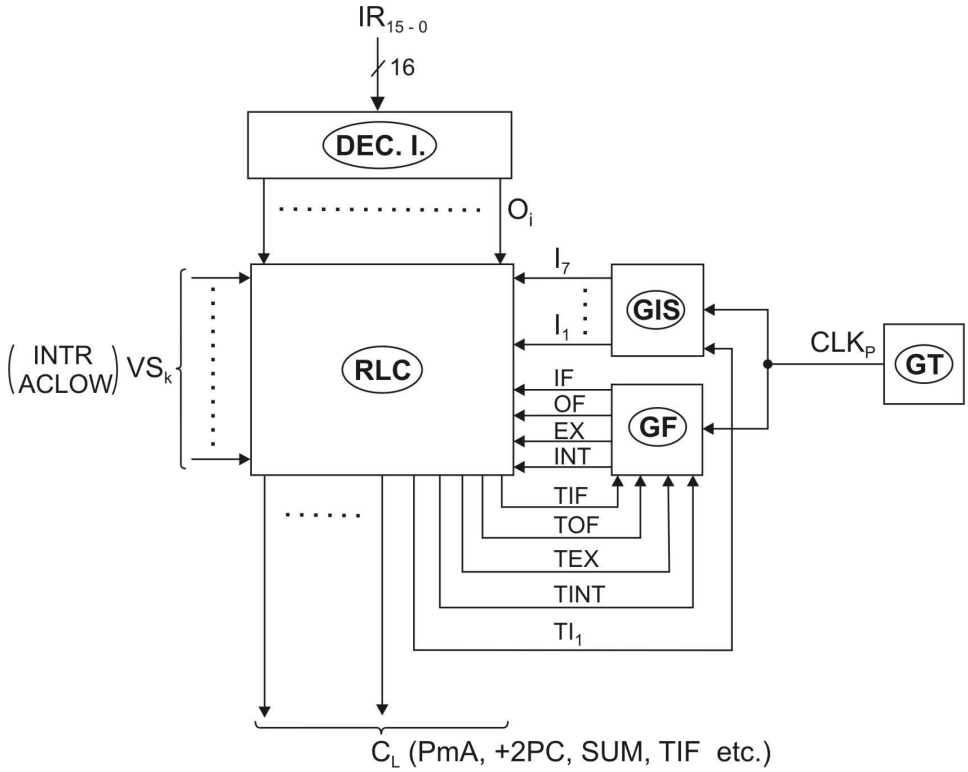
În variantă cablată, BGC conţine trei elemente în structura sa (fig 2.83):

1. RLC - reţeaua logică combinaţională (structură pur combinaţională)
2. GF - generatorul de faze (structură secvenţială)
3. GIS - generatorul impulsurilor de stare (structură secvenţială)

Dacă inventariem tabelele de comenzi, constatăm că numărul maxim de impulsuri de stare utilizate este 7 (faza INT, figurile 2.71 şi 2.72). Rezultă că GIO trebuie să genereze 7 impulsuri de stare.

Ciclul instrucţiunii conţine 3 faze (IF, OF şi EX) pentru instrucţiunile din clasa IOP şi respectiv 2 faze (IF şi EX) pentru instrucţiunile din clasa ID (figura 2.12). Dacă pe durata procesării unei instrucţiuni se activează cererea de întrerupere INTR, la sfârşitul instrucţiunii (în punctul interuptibil) se intră în faza de întrerupere INT; în caz contrar se revine în faza IF pentru *fetch*-ul următoarei instrucţiuni din program (figura 2.85). Trecerea din faza curentă în faza următoare se realizează prin comenzile *TIF*, *TOF*, *TEX* şi respectiv *TINT*.

Fiecare fază conţine unul au mai multe impulsuri de stare (numărul maxim de impulsuri este 7 şi apare în faza INT care este cea mai complexă). Succesiunea impulsurilor de stare în cadrul fazelor este obligatorie: $I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow \ldots$



DEC. I. -decodificatorul instrucţiunii
RLC -reţeaua logică combinaţională
GF -generatorul de faze
GT -generatorul de tact
GIS -generatorul impulsurilor de stare
$IR_{15 \div 0}$ -codul instrucţiunii curente (conţinutul registrului IR)
O_i -semnale care specifică operaţia (instrucţiunea) în curs de execuţie. Pentru fiecare instrucţiune încărcată în IR se decodifică *OPCODE*-ul şi (la instrucţiunile din clasa IOP) suplimentar şi modul de adresare (MA). Semnalele din clasa O_i rezultate prin decodificarea *OPCODE*-ului sunt: LDA, LDSP, … , IRET iar cele rezultate prin decodificarea câmpului MA sunt: AM, AD, AI şi AX.
VS_k -variabile de stare provenite din exteriorul BGC-ului (INTR, ACLOW)
C_L -comenzi logice elementare
IF, OF, EX, INT -semnale de fază (generate de către GF)
$I_1 \div I_7$ -impulsuri de stare (generate de către GIS)
CLK_P -semnalul detact al procesorului

Fig 2.83 Structura unităţii de control cablate

După încheierea fazei curente se intră în I_1 aferent fazei următoare (comanda TI_1 pentru GIS realizează trecerea în I_1). Dacă analizăm tabelele de comenzi aferente diverselor faze şi respectiv diverselor instrucţiuni, constatăm că TI_1 se activează întotdeauna în tandem cu o comandă de tipul "*treci în faza următoare*". Rezultă că, ori de câte ori GF-ul trece dintr-o fază curentă într-o fază următoare, se va produce şi iniţializarea GIS-ului (trecerea GIS-ului în I_1).

Fazele nu au o lungime standard. Faza EX, spre exemplu, conţine 4 impulsuri de stare ($I_1 \div I_4$) în cazul instrucţiunii CALL *adr*, un impuls de stare (I_1) în cazul instrucţiunii LD A, *src*, 2 impulsuri (I_1 şi I_2) în cazul instrucţiunii RET, etc.

GIS şi GF sunt blocuri secvenţiale comandate de către generatorul de tact (GT), prin semnalul de tact CLK_P. GF generează semnalele de fază (asignate celor 4 faze definite: IF, OF, EX şi INT) iar GIS generează impulsurile de stare ($I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow \ldots$) asignate perioadelor de tact din componenţa fiecărei faze.

Pe ieşirile RLC apar toate comenzile logice C_L cuprinse în tabelele de comenzi elaborate. Comenzile C_L le numim comenzi elementare deoarece declanşează operaţii elementare (operaţii executabile într-o singură perioadă de tact). Expresia unei astfel de comenzi se obţine pe baza tabelelor de comenzi şi va avea forma generală:

$$C_{Lj} = \sum_{i,j,k,m} O_i \cdot VS_j \cdot F_k \cdot I_m \qquad (2.11)$$

Ecuaţia 2.11 conţine o sumă de produse logice în care apar următorii termeni:

O_i -tipul instrucţiunii din IR, modul de adresare, etc. (semnale obţinute prin decodificarea codului instrucţiunii curente)

VS_j -variabile de stare provenite din exteriorul unităţii de contrul (ACLOW, INTR)

F_k -faza curentă (IF, OF, EX, INT)

I_m -impulsul de stare din cadrul fazei curente (I_1, I_2, ….)

Din comanda de tip nivel C_{Lj}, acolo unde este cazul, se va obţine comanda de tip impuls C^*_{Lj} (aşa cum indică figura 2.84)

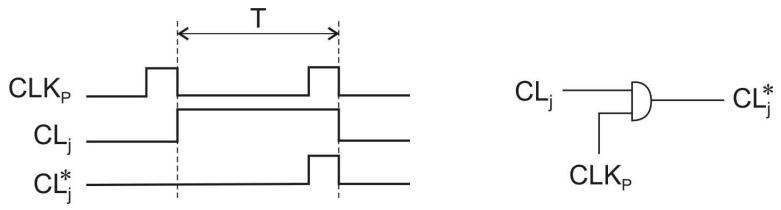


Fig 2.84 Generarea comenzii de tip impuls din comanda de tip nivel corespondentă (CLK_P –semnalul de tact al procesorului)

Pe baza tabelelor de comenzi elaborate, se vor scrie ecuaţiile tuturor comenzilor C_L. Se va obţine astfel un sistem de ecuaţii logice care descrie complet reţeaua RLC. Proiectarea reţelei RLC se face pornind de la acest sistem de ecuaţii logice.

2.10.1 Generatorul de faze (GF)

Pentru că vizăm o proiectare bine structurată şi respectiv sistematizată, am descompus ciclul instrucţiunii în subcicluri pe care le-am denumit faze. Succesiunea fazelor în timpul procesării instrucţiunilor, explicată mai detaliat la 2.4, este redată şi în figura 2.85:

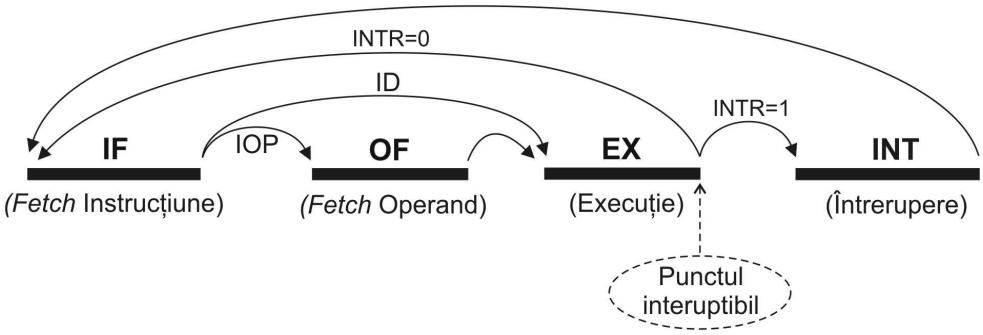


Fig 2.85 Succesiunea fazelor în timpul procesării instrucţiunilor

În figura 2.85 apar patru faze; trei sunt cuprinse în ciclul instrucţiunii (IF, OF şi EX) iar cea de-a patra (INT) se parcurge doar la activarea unei întreruperi. Rezultă că GF va genera 4 semnale de fază, care pot fi codificate cu ajutorul a 2 bistabili (**B0**, **B1**). Tabelul 2.8 prezintă codificarea pe care ne-o propunem.

O implementare posibilă a generatorului de faze, care utilizează doi bistabili *J-K master-slave*, este redată în figura 2.86. Această impementare respectă codificarea fazelor din tabelul 2.8

Starea		Faza
B1	B0	
0	0	IF
0	1	OF
1	1	EX
1	0	INT

Tabelul 2.8 Codificarea fazelor

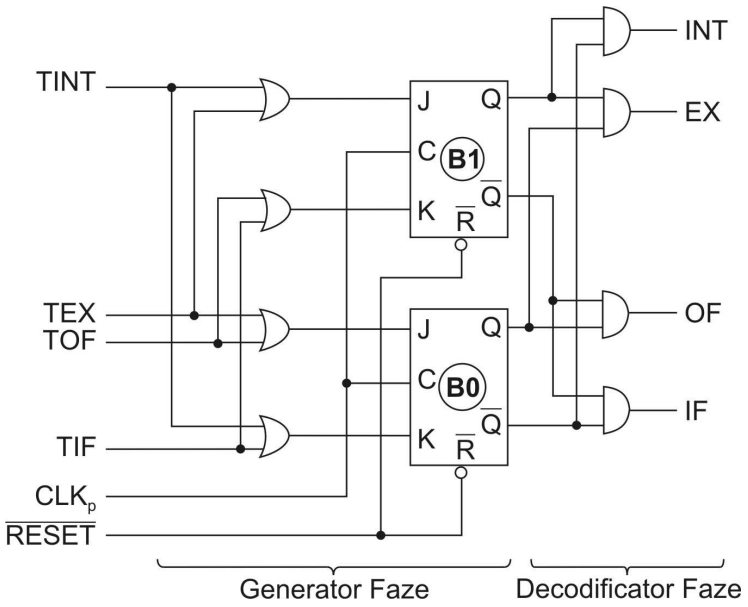


Fig 2.86 Generatorul de faze (GF)
(CLK_P –semnalul de tact al procesorului)

La iniţializarea sistemului ($\overline{RESET} = 0$), prin resetarea asincronă a celor doi bistabili (B1 şi respectiv B0), se instalează faza IF. Pe măsură ce instrucţiunea avansează în execuţie, cu ajutorul comenzilor TIF, TOF, TEX şi TINT, se face trecerea din faza curentă în faza următoare. Aceste comenzi se obţin pe ieşirile RLC şi sunt dedicate generatorului de faze GF. În toate tabelele de comenzi elaborate pentru diversele instrucţiuni şi respectiv faze, cele patru comenzi (TIF, TOF, TEX şi TINT) apar întotdeauna pe ultima coloană şi sunt întotdeauna însoţite de comanda TI_1 (comanda de iniţializare GIS). Aceasta înseamnă că, odată cu următorul impuls de tact procesor, GF va trece în faza următoare şi respectiv GIS va trece în I_1. **Asta e, fiecare nouă fază începe cu impulsul de stare I_1 (care marchează prima perioadă de tact din cadrul fazei)!**

Decodificatorul de faze din figura 2.86 (format din cele patru porti SI) decodifică starea GF şi generează semnalele de fază asociate.

Să remarcăm faptul că în figura 2.86, cele patru comenzi (TIF, TOF, TEX şi TINT), se aplică (prin intermediul unor porţi SAU) pe intrările de setare/resetare sincronă (intrările J şi respectiv K) aferente celor doi bistabili din structura GF (bistabilii B1 şi respectiv B2). Asta înseamnă că cei doi bistabili vor fi setaţi/resetaţi sincron de cele patru comenzi (TIF, TOF, TEX şi TINT). Cu alte cuvinte, setarea/resetarea va fi realizată în sincronism cu impulsurile de tact CLK_P. Frontul urcător al impulsului de tact (CLK_P) provoacă setarea bistabilului *master* iar cel coborâtor va transfera starea bistabilului *master* în bistabilul *slave* (a se revedea structura şi funcţionarea bistabilului *JK master-slave*). Prin urmare, comutarea ieşirilor GF-ului (comutarea semnalelor de fază) se va face sincron cu frontul coborâtor al tactului CLK_P (aşa cum indică cronograma de semnale din figura 2.87). În figura 2.87 sunt reprezentate două comutări succesive de faze: comutare din EX în IF urmată de comutare

din IF în OF. Se poate observa că momentul comutării coincide cu momentul trecerii dintr-o perioadă de tact (perioada de tact curentă) într-o altă perioadă de tact (perioada de tact următoare). Comutarea se produce numai dacă, pe perioada de tact curentă, se activează comanda *„treci în faza următoare”*. Prin sintagma *„treci în faza următoare”* înţelegem desigur una dintre comenzile: TIF, TOF, TEX sau TINT.

Faptul că, în tabelele de comenzi, pe ultima coloană (pe ultimul impuls de stare) am codificat întotdeauna o comandă *„treci în faza următoare”*, face ca implementarea din figura 2.86 să corespundă cerinţelor (fixate evident chiar prin tabelele de comenzi).

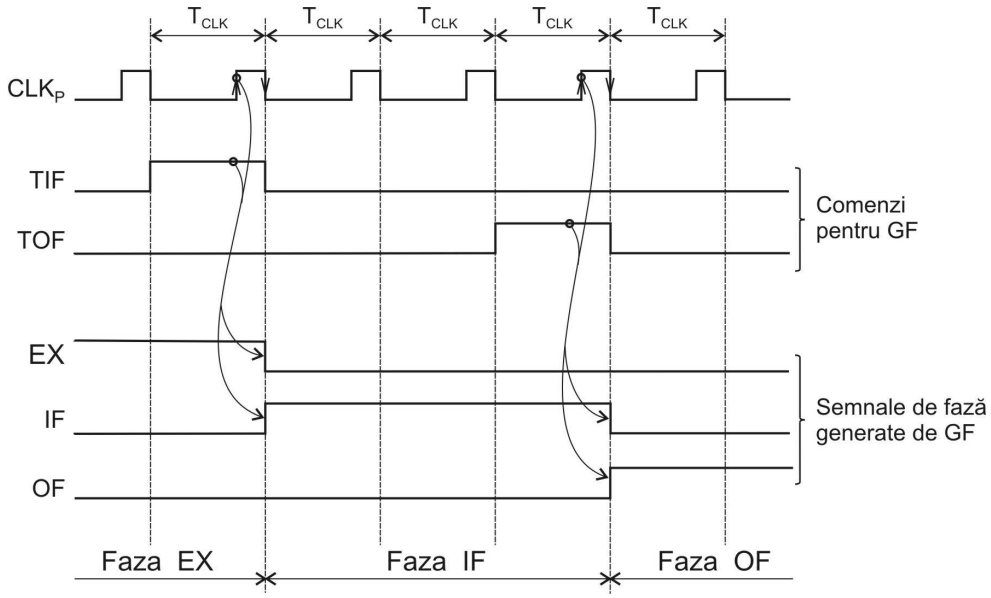


Fig 2.87 Două comutări succesive de faze (din EX în IF şi respectiv din IF în OF)

2.10.2 Generatorul impulsurilor de stare (GIS)

Cea mai complexă fază s-a dovedit a fi faza de întrerupere care conţine 7 impulsuri de stare ($I_1 \div I_7$). Prin urmare, GIS trebuie să genereze 7 impulsuri de stare

O variantă simplă de implementare în *hardware* a GIS se bazează pe un registru de deplasare în inel cu 7 bistabili *JK-master-slave* (figura 2.88)

Cu $\overline{RESET} = 0$ (la iniţializarea procesorului) GIS este adus în starea iniţială (I_1=1; I_2=I_3=…=I_7=0). În continuare cu fiecare impuls de tact procesor (CLK_P) starea bistabilului B_i este transferată în bistabilul B_{i+1} iar starea bistabilului B_7 este rebuclată pe intrările bistabilului B_1.

Deficienţa acestei implementări constă în faptul că generează faze de lungime fixă (faze care se întind în mod constant pe 7 perioade de tact procesor). Cronograma de semnale care ilustrează funcţionarea unui astfel de GIS (cu faze de lungime fixă) este redată în figura 2.89

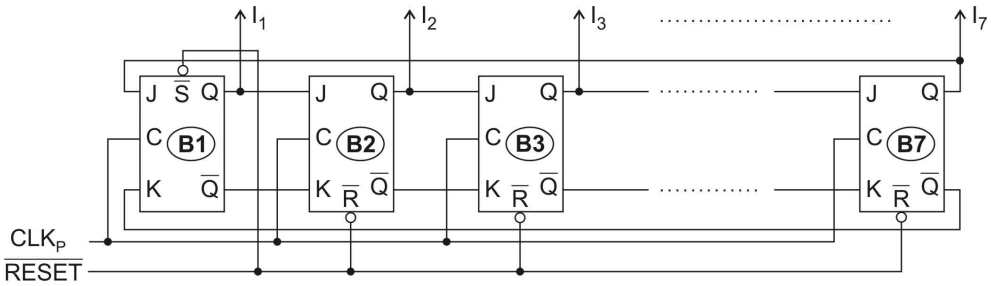


Fig 2.88 GIS bazat pe un registru de deplasare în inel
($\overline{S}$, $\overline{R}$ -intrări de setare/resetare asincrone)

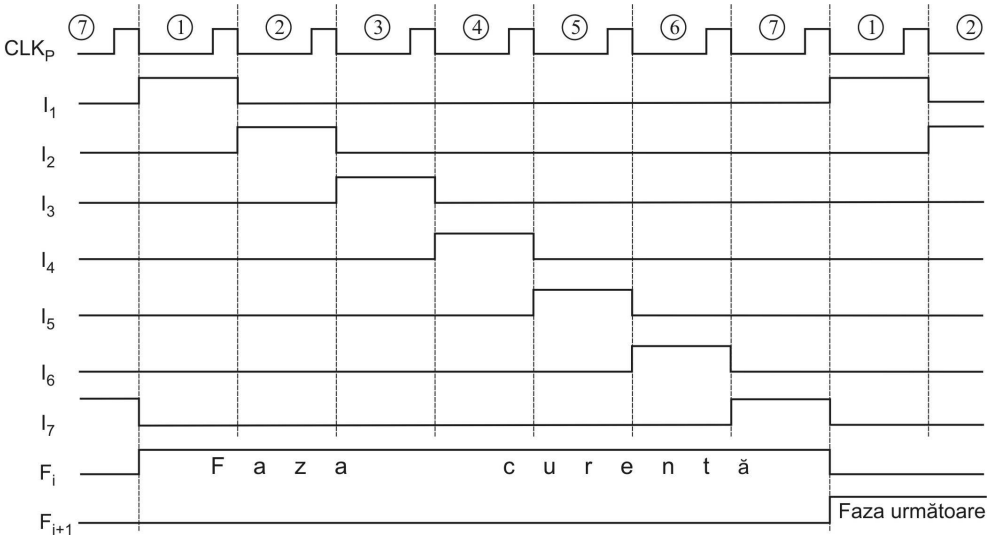


Fig 2.89 Cronograma impulsurilor de stare aferentă GIS cu faze de lungime fixă

Schema GIS cu faze de lungime fixă este deficitară deci din punctul de vedere al performanţelor obtenabile. Pentru creşterea vitezei de procesare se impune generarea fazelor de lungime variabilă. Fiecare fază trebuie să conţină doar impulsurile de stare utilizate în generarea de comenzi C_L (vezi tabelele de comenzi). Altfel spus, fiecare fază trebuie să aibe lungimea care i-a fost stabilită prin tabelul de comenzi asociat

Exemplu: Faza IF trebuie să conţină 3 impulsuri de stare (lungimea stabilită prin tabelul 2.16), faza OF trebuie să conţină 2 impulsuri (lungimea stabilită prin tabelul 2.20), faza EX aferentă instrucţiunii LD A, *src* va conţine un singur impuls (lungimea stabilită prin tabelul generic 2.23), etc.

Un GIS cu faze de lungime variabilă se poate obţine, introducând comanda TI_1 („treci în I_1”) în schema GIS-ului cu faze de lungime fixă din figura 2.88. Rezultatul va fi desigur un GIS cu faze de lungime variabilă (figura 2.90) care ne asigură că fiecare fază va avea durata stabilită prin tabelul de comenzi aferent (fiecare fază va conţine doar impulsurile de stare cuprinse în tabelul de comenzi aferent).

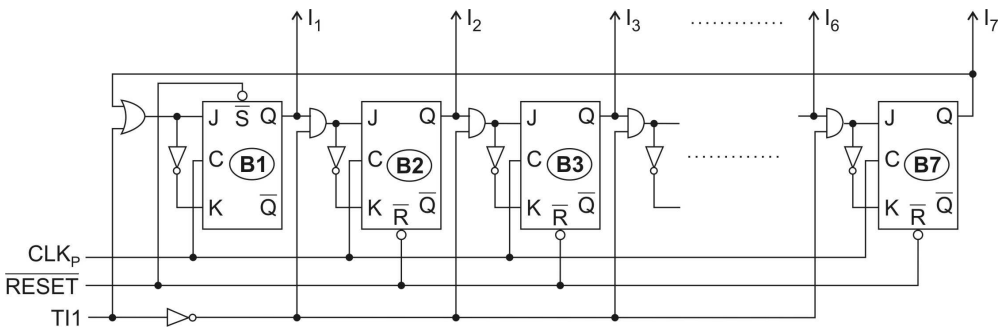


Fig 2.90 Structura GIS cu faze de lungime variabilă

În schema GIS cu faze de lungime variabilă, comanda TI_1 este în fond o comandă de iniţializare. Prin urmare:

- Dacă TI_1=0 (inactivă) ⇒ GIS trece din I_i în I_{i+1}, pe fiecare impuls de tact CLK_P. Din I_7 ar urma să se revină în I_1. În această manieră se parcurg impulsurile de stare din cadrul unui tabel de comenzi.

- Dacă TI_1=1 (activă) ⇒ GIS trece din I_i (pentru orice i) în I_1. Cu alte cuvinte, comanda TI_1 provoacă iniţializarea GIS. În această manieră se trece din ultimul impuls de stare aferent fazei curente, în primul impuls aferent fazei următoare.

Comanda TI_1 apare (în fiecare tabel de comenzi) pe ultimul impuls de stare utilizat la generare de comenzi C_L. Ea este întotdeauna asociată cu o comandă TF_j (“*treci în faza următoare*”). În consecinţă, BGC va trece concomitent GF în faza următoare şi respectiv GIS în I_1. Prin urmare, fiecare fază începe cu I_1 şi are o lungima bine definită. Faza se încheie cu impulsul de stare în care se activează comanda TF_j (activată întotdeauna concomitent cu TI_1). Pe următoarea perioadă de tact, GF va fi în faza următoare iar GIS va fi în I_1.

Să remarcăm faptul că (figura 2.90), atunci când se activează, comanda TI_1 realizează J=1/K=0 la bistabilul B1 şi respectiv J=0/K=1 la bistabilii următori (B2÷B7). Prin urmare, trecerea GIS-ului în I_1 (iniţializarea GIS-ului) este o operaţie sincronă (este realizată de impulsul CLK_P din starea respectivă). Ca şi GF-ul, GIS-ul este implementat cu bistabili *JK master-slave*. Prin urmare, frontul urcător al impulsului de tact (CLK_P) încarcă

starea intrărilor (J şi respectiv K) în bistabilul *master* iar frontul coborâtor transferă starea bistabilului *master* în bistabilul *slave*. Asta înseamnă că, activarea/dezactivarea ieşirilor GIS-ului (comutarea impulsurilor de stare) se face sincron cu frontul coborâtor al semnalului de tact CLK_P (ca în figura 2.89). Prin urmare, implementarea din figura 2.90 satisface cerinţele impuse:

- faze de lungime variabilă
- comutarea impulsurilor de stare se face sincron cu frontul coborâtor al tactului procesorului (CLK_P).

2.10.3 Reţeaua Logică Combinaţională (RLC)

Reţeaua RLC (figura 2.91) împreună cu GF şi GIS formează unitatea de control a procesorului. Unitatea de control generează comenzile logice elementare (C_L) pe care le regăsim în tabele de comenzi aferente diverselor faze/instrucţiuni, (tabelele elaborate pe parcursul paragrafelor 2.5÷2.8). Din acest motiv, **unitatea de control** este denumită deseori **bloc de generare comenzi** (BGC).

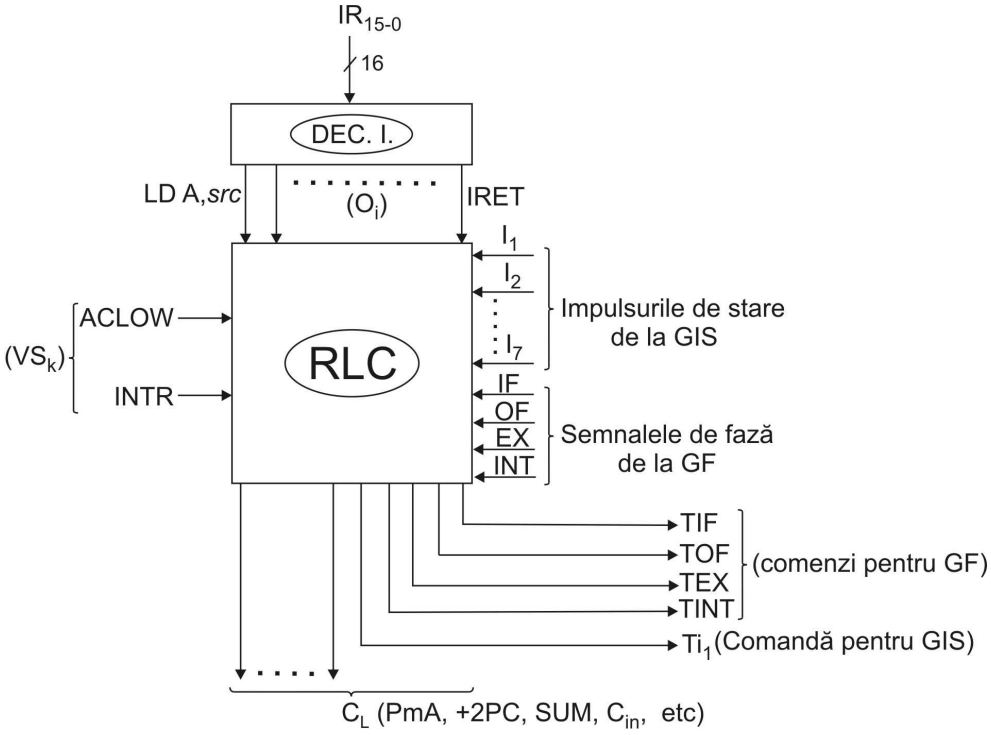


FIG 2.91 Reţeaua logică combinaţională (RLC)

GF şi GIS sunt blocuri secvenţiale care marchează timpul (marchează perioadele de tact cuprinse în ciclul instrucţiunii). Cu alte cuvinte, GF împarte (grosier) ciclul instrucţiunii în faze (pe care le marchează cu semnalele de fază) iar GIS împarte (fin) fazele în perioade de tact (pe care le marchează cu impulsuri de stare). Pe fiecare impuls de stare (care face parte evident dintr-o anumită fază) trebuie activat un set de comenzi elementare (C_L). Aceste seturi de comenzi sunt bine precizate şi această precizare este realizată cu ajutorul tabelelor de comenzi.

Pentru efectuarea unei operaţii elementare (operaţia elementară este operaţia efectuată pe o singură perioadă de tact) trebuie activat un set de comenzi elementare (vezi 2.9.4 unde este descrisă operaţia A←A+M). Există şi operaţii declanşate de o singură comandă. De exemplu, operaţia PC←PC+2 este efectuată la activarea comenzii *+2PC*.

Reţeaua RLC are pe intrări următoarele variabile (semnale):

- O_i -variabile care precizează tipul instrucţiunii din IR, modul de adresare, etc. (obţinute prin decodificarea codului instrucţiunii curente).
- ACLOW, INTR -variabile de stare externe care precizează dacă au apărut situaţii deosebite în timpul procesării instrucţiunii curente.
- IF, OF, EX, INT -semnalele de fază (generate de GF) care precizează faza în care s-a ajuns, în timpul procesării instrucţiunii curente.
- $I_1 \div I_7$ -impulsurile de stare (generate de GIS) care precizează perioada de tact la care s-a ajuns, în cadrul fazei curente.

Pe baza acestor semnale de intrare, reţeaua RLC trebuie să genereze toate comenzile elementare (C_L) care apar în diversele tabele de comenzi. Funcţionarea acestei reţele pur cumbinaţionale este descrisă de un sistem complex de ecuaţii logice. Acest sistem se obţine printr-o metodologie foarte simplă, care porneşte de la tabelele de comenzi. Subliniem faptul că **trebuie luate în considerare toate tabelele de comenzi**.

Sistemul va conţine tot atâtea ecuaţii, câte comenzi distincte putem identifica în tabelele de comenzi. Deci, într-o primă etapă, trebuie să obţinem **lista tuturor comenzilor**, pe care le putem identifica în tabelele de comenzi. Este evident o simplă problemă de inventariere a comenzilor.

În etapa a doua trebuie să obţinem, succesiv, **ecuaţiile tuturor comenzilor din listă**. Elaborarea ecuaţiei aferente unei anumite comenzi presupune o nouă etapă de inventariere. Trebuie inventariate toate apariţiile respectivei comenzi, în tabelele de comenzi, deoarece fiecare apariţie va induce un produs logic în ecuaţia respectivei comenzi.

Prin urmare, ecuaţia unei anumite comenzi va conţine o sumă de produse logice. Apariţia comenzii respective, pe o anumită coloană a unui tabel de comenzi, generează un produs logic în ecuaţia comenzii. Prin urmare, numărul de produse logice pe care le va conţine ecuaţia unei comenzi, va fi egal cu numărul de apariţii ale respectivei comenzi în tabelele de comenzi. Subliniem încă o dată, **trebuie luate în considerare toate tabelele de comenzi**.

Ecuaţiile tuturor comenzilor din listă formează un **sistem de ecuaţii logice** iar sistemul defineşte complet structura reţelei RLC. Pentru edificare, vom parcurge algoritmul de elaborare a câtorva ecuaţii:

1. Comanda *PdPC* apare:

- în faza IF, pe impulsul I_1. Această apariţie generează produsul $IF \cdot I_1$
- în faza EX aferentă instrucţiunii **JMP *adr*** cu adresare imediată (adică dacă AM=1), pe impulsul I_1. Acestei apariţii îi va corespunde, în ecuaţie, produsul $JMP \cdot EX \cdot AM \cdot I_1$.
- în faza EX aferentă instrucţiunilor de salt condiţionat cu adresare imediată (adică dacă AM=1), pe impulsul I_1. Cele 8 apariţii din tabelele aferente instrucţiunilor de salt condiţionat vor genera deci 8 produse logice, aşa cum indică tabelul 2.9.

Instrucţiune	Produsul generat
JC	$JC \cdot EX \cdot AM \cdot I_1$
JNC	$JNC \cdot EX \cdot AM \cdot I_1$
JZ	$JZ \cdot EX \cdot AM \cdot I_1$
JNZ	$JNZ \cdot EX \cdot AM \cdot I_1$
JP	$JP \cdot EX \cdot AM \cdot I_1$
JM	$JM \cdot EX \cdot AM \cdot I_1$
JV	$JV \cdot EX \cdot AM \cdot I_1$
JNV	$JNV \cdot EX \cdot AM \cdot I_1$

Tabelul 2.9 Cele opt produse logice induse în ecuaţia comenzii *PdPC* de faza EX aferentă celor opt instrucţiuni de salt condiţionat

- în faza EX aferentă instrucţiunii **CALL *adr***. În acest tabel de comenzi identificăm două apariţii ale comenzii *PdPC*:
  - pe impulsul I_3. Acestă apariţie generează produsul $CALL \cdot EX \cdot I_3$.
  - pe impulsul I_4, dacă AM=1. Acestă apariţie generează produsul $CALL \cdot EX \cdot AM \cdot I_4$.
- în faza INT, pe impulsul I_5; această apariţie generează produsul $INT \cdot I_5$.

Prin urmare, ecuaţia comenzii *PdPC* va conţine toate aceste produse logice:

$$\begin{aligned} PdPC = {} & IF \cdot I_1 + JMP \cdot EX \cdot AM \cdot I_1 + JC \cdot EX \cdot AM \cdot I_1 + JNC \cdot EX \cdot AM \cdot I_1 + \\ & + JZ \cdot EX \cdot AM \cdot I_1 + JNZ \cdot EX \cdot AM \cdot I_1 + JP \cdot EX \cdot AM \cdot I_1 + JM \cdot EX \cdot AM \cdot I_1 + \\ & + JV \cdot EX \cdot AM \cdot I_1 + JNV \cdot EX \cdot AM \cdot I_1 + CALL \cdot EX \cdot I_3 + CALL \cdot EX \cdot AM \cdot I_4 + INT \cdot I_5 \end{aligned} \tag{2.11}$$

2. Comanda *SUM* apare:

- în faza OF, pe impulsul I_1, dacă AX=1 (dacă instrucţiunea curentă utilizează adresarea indexată). Această apariţie generează produsul $OF \cdot AX \cdot I_1$.
- în faza EX a instrucţiunii **ADD A,*src***, pe I_1. Această apariţie generează produsul $ADD \cdot EX \cdot I_1$.
- în faza EX a instrucţiunii **SUB A,*src***, pe I_1. Această apariţie generează produsul $SUB \cdot EX \cdot I_1$.
- în faza EX a instrucţiunii **CMP A,*src***, pe I_1. Această apariţie generează produsul $CMP \cdot EX \cdot I_1$.
- în faza EX a instrucţiunii **INC *src***, pe I_1. Această apariţie generează produsul $INC \cdot EX \cdot I_1$.
- în faza EX a instrucţiunii **DEC *src***, pe I_1. Această apariţie generează produsul $DEC \cdot EX \cdot I_1$.
- în faza EX aferentă instrucţiunii **JMP *adr***, pe I_1, dacă AM=1. Acestei apariţie generează produsul $JMP \cdot EX \cdot AM \cdot I_1$.
- în faza EX aferentă instrucţiunilor de salt condiţionat cu adresare imediată (adică dacă AM=1!), pe impulsul I_1. Cele 8 apariţii din tabelele aferente instrucţiunilor de salt condiţionat vor genera deci 8 produse logice; vor fi chiar produsele logice din tabelul 2.9 (care au apărut şi la comanda *PdPC*).
- în faza EX aferentă instrucţiunii **CALL *adr***, pe I_4, dacă AM=1 sau dacă AX=1. Cele două apariţii generează produsele $CALL \cdot EX \cdot AM \cdot I_4$ şi respectiv $CALL \cdot EX \cdot AX \cdot I_4$.
- în faza EX a instrucţiunii **INC A**, pe I_1. Această apariţie generează produsul $INCA \cdot EX \cdot I_1$.
- în faza EX a instrucţiunii **DEC A**, pe I_1. Această apariţie generează produsul $DECA \cdot EX \cdot I_1$.

Prin urmare, ecuaţia comenzii *SUM* va conţine toate produse logice enumerate:

$$\begin{aligned} SUM = {} & OF \cdot AX \cdot I_1 + ADD \cdot EX \cdot I_1 + SUB \cdot EX \cdot I_1 + CMP \cdot EX \cdot I_1 + INC \cdot EX \cdot I_1 + \\ & + DEC \cdot EX \cdot I_1 + JMP \cdot EX \cdot AM \cdot I_1 + JC \cdot EX \cdot AM \cdot I_1 + JNC \cdot EX \cdot AM \cdot I_1 + \\ & + JZ \cdot EX \cdot AM \cdot I_1 + JNZ \cdot EX \cdot AM \cdot I_1 + JP \cdot EX \cdot AM \cdot I_1 + JM \cdot EX \cdot AM \cdot I_1 + \\ & + JV \cdot EX \cdot AM \cdot I_1 + JNV \cdot EX \cdot AM \cdot I_1 + CALL \cdot EX\ AM \cdot I_4 + \\ & + CALL \cdot EX \cdot AX \cdot I_4 + INCA \cdot EX \cdot I_1 + DECA \cdot EX \cdot I_1 \end{aligned} \tag{2.12}$$

3. Comanda *TEX („treci în faza de execuţie”)* apare:

- în faza IF, pe impulsul I_3, dacă ID=$\overline{IOP}$ =1 (dacă instrucţiunea curentă este din clasa ID). Această apariţie generează produsul $IF \cdot ID \cdot I_3$.
- în faza OF, pe impulsul I_1, dacă AM=1. Această apariţie generează produsul $OF \cdot AM \cdot I_1$.
- în faza OF, pe impulsul I_1, dacă AD=1. Această apariţie generează produsul $OF \cdot AD \cdot I_1$.
- în faza OF, pe impulsul I_2. Această apariţie generează produsul $OF \cdot I_2$.

Prin urmare, ecuaţia comenzii *TEX* va conţine toate produse logice enumerate:

$$TEX = IF \cdot ID \cdot I_3 + OF \cdot AM \cdot I_1 + OF \cdot AD \cdot I_1 + OF \cdot I_2 \qquad (2.13)$$

4. Comanda *+2SP („incrementare cu 2 a pointerului de stivă”)* apare:

- în faza EX a instrucţiunii **POP *reg***, pe I_2. Această apariţie generează produsul $POP \cdot EX \cdot I_2$.
- în faza EX a instrucţiunii **POP A**, pe I_2. Această apariţie generează produsul $POPA \cdot EX \cdot I_2$.
- în faza EX a instrucţiunii POP FLAG, pe I_2. Această apariţie generează produsul $POPFLAG \cdot EX \cdot I_2$.
- în faza EX a instrucţiunii **RET**, pe I_2. Această apariţie generează produsul $RET \cdot EX \cdot I_2$.
- în faza EX a instrucţiunii **IRET**, pe I_2 şi respectiv pe I_4. Cele două apariţii generează produsele $IRET \cdot EX \cdot I_2$ şi respectiv $IRET \cdot EX \cdot I_4$

Prin urmare, ecuaţia comenzii *+2SP* va conţine toate produse logice enumerate:

$$\begin{aligned} +2SP = {} & POP \cdot EX \cdot I_2 + POPA \cdot EX \cdot I_2 + POPFLAG \cdot EX \cdot I_2 + \\ & + RET \cdot EX \cdot I_2 + IRET \cdot EX \cdot I_2 + IRET \cdot EX \cdot I_4. \end{aligned} \qquad (2.14)$$

În această manieră, se va parcurge întreaga listă de comenzi, elaborată pe baza tabelelor de comenzi, (în prima etapă) şi se vor elabora (în a doua etapă) ecuaţiile tuturor comenzilor din listă. Toate aceste ecuaţii formează un sistem amplu de ecuaţii logice iar sistemul defineşte complet structura şi fincţionarea reţelei RLC.

Implementarea în *hardware* a reţelei RLC este foarte simplă. Pentru fiecare ecuaţie se va obţine o implementare cu porţi logice, ierarhizată pe două nivele: un prim nivel de porţi SI, urmat de o poartă SAU pe al doilea nivel.

Exemple:

1. Comanda *PdPC* va fi implementată în *hardware*, pornind de la ecuaţia 2.11. Cele 13 produse logice din ecuaţia 2.11 revendică 13 porţi SI (pe primul nivel) iar suma acestor produse revendică o poartă SAU (cu 13 intrări), pe al doilea nivel. Schema obţinută este redată în figura 2.92.

2. Comanda *SUM* va fi implementată în *hardware*, pornind de la ecuaţia 2.12, care conţine 19 produse logice. Se poate observa că 10 dintre cele 19 produse le regăsim şi în ecuaţia 2.11 (ecuaţia comenzii *PdPC*). Soluţia de implementare în *hardware* poate să ţină seama de faptul că cele 10 produse logice apar în ecuaţiile ambelor comenzi (sunt produse comune) şi drept consecinţă, implementarea poate fi simplificată . Prin urmare, în schema de generare a comenzilor *PdPC* şi *SUM* din figura 2.93 apar doar 22 porţi SI, deoarece 10 porţi sunt utilizate în comun. Cele 10 porţi utilizate în comun implementează evident cele 10 produse logice comune.

   Să remarcăm faptul că această reducere de complexitate nu conduce şi la degradarea performanţelor. Întârzierea indusă în generarea comenzilor *PdPC* şi *SUM* (întârzierea t_{gc} din figura 2.82) rămâne minimă (egală cu timpul de propagare peste două nivele logice, primul nivel fiind reprezentat de porţile SI iar al doilea de porţile SAU). Prin urmare, orice simplificare a *hardware*-ului, care nu atrage drept consecinţă degradarea performanţelor, este binevenită.

3. Pentru implementarea în *hardware* a comenzii *TEX* se va utiliza ecuaţia 2.13 iar pentru implementarea comenzii *+2SP*, ecuaţia 2.14. Schemele obţinute sunt redate în figurile 2.94.a). şi respectiv 2.94.b).

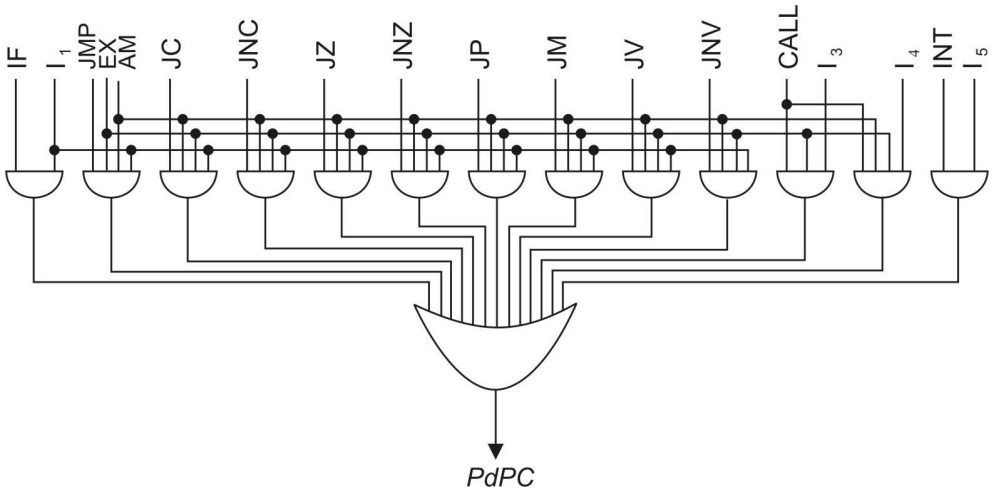


Fig 2.92 Implementarea comenzii PdPC în hardware

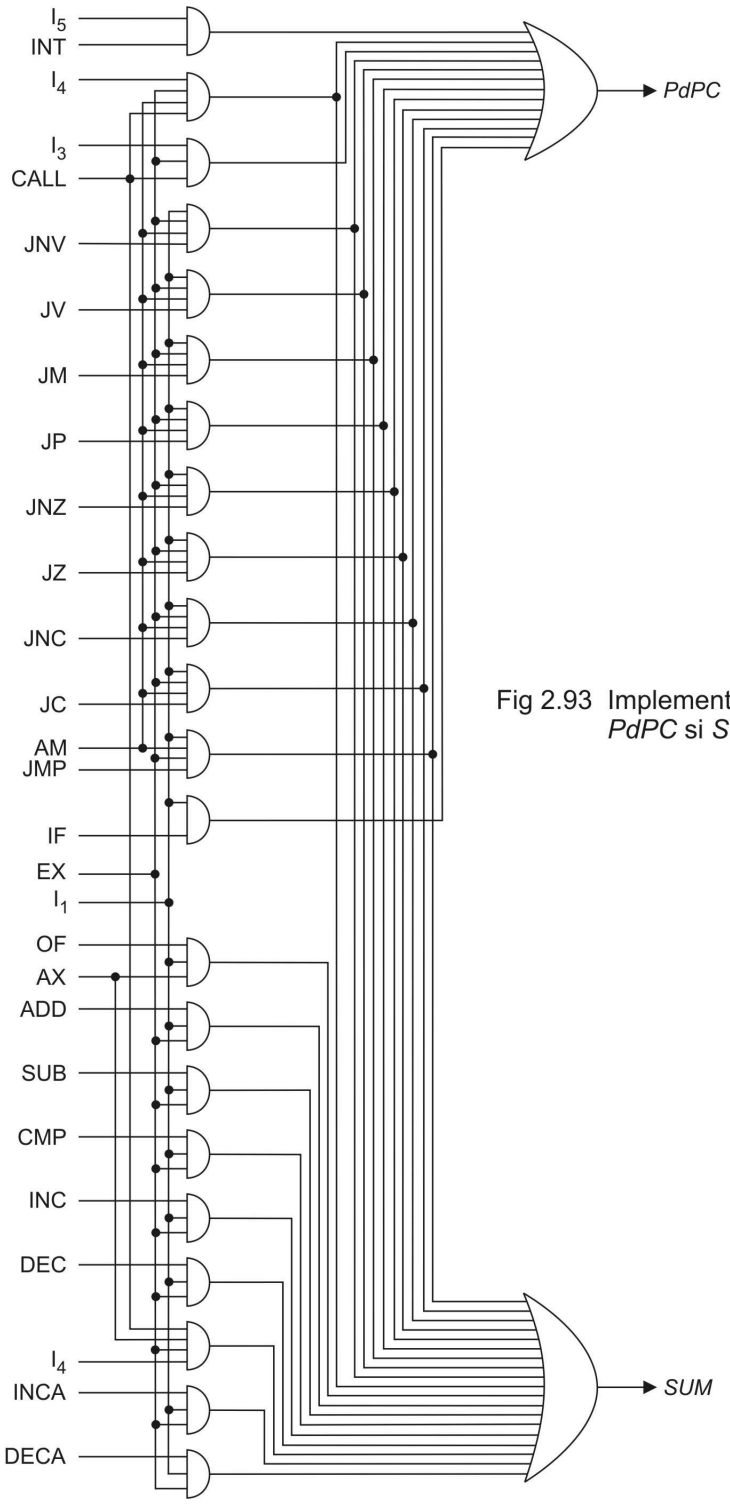


Fig 2.93 Implementarea comenzilor *PdPC* si *SUM* în *hardware*

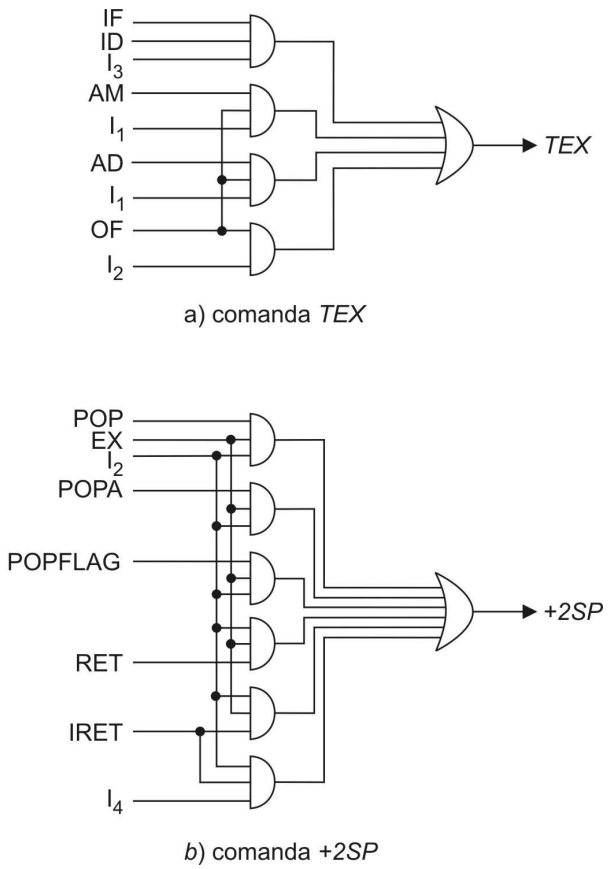


Fig 2.94 Implementarea comenzilor *TEX* şi *+2SP*

Implementând în această manieră ecuaţiile tuturor comenzilor elementare (C_L), se va obţine reţeaua RLC implementată cu circuite logice combinaţionale (reţeua RLC cablată). Întârzierea t_{gc} pe care reţeaua RLC o va induce în generarea comenzilor elementare (C_L) va fi identică pentru toate comenzile. Aceast **timp de întârziere** (denumit **timp de generare comandă** (t_{gc}) în figura 2.82) va fi egal cu timpul de propagare peste două nivele logice, pe primul nivel situându-se întotdeauna o poartă SI iar pe al doilea o poartă SAU.

În concluzie, în variantă cablată (aşa cum se implementează la procesoarele RISC), unitatea de control a procesorului (blocul BGC) conţine trei elemente în structura sa (fig 2.83):

1. RLC -reţeaua logică combinaţională (structură pur combinaţională)

2. GF -generatorul de faze (structură secvenţială)

3. GIS -generatorul impulsurilor de stare (structură secvenţială)

Generatorul de faze (GF) împarte ciclul instrucţiunii în subcicluri (faze), asociind fiecărei faze un semnal de fază (IF, OF, EX şi respectiv INT). GF este un circuit secvenţial (proiectat la 2.10.1).

GIS împarte fiecare fază în impulsuri de stare iar impulsurile de stare identifică perioadele de tact din cadrul fiecărei faze. Pe fiecare perioadă de tact (impuls de stare) se efectuează una sau mai multe operaţii elementare (concurente). Între operaţiile concurene nu trebuie să apară conficte. (vezi explicaţiile de la 2.4.A). Ca şi GF, GIS este un circuit secvenţial (proiectat la 2.10.2).

Reţeua RLC (proiectată la 2.10.3) este un circuit pur combinaţional, proiectat pe baza unui sistem de ecuaţii logice. Sistemul de ecuaţii s-a obţinut cu ajutorul tabelelor de comenzi.

Această structurare a unităţii de control în 3 blocuri distincte, bine delimitate din punct de vedere funcţional, ne-a facilitat o abordare clară, bine etapizată şi foarte bine structurată a activităţii de proiectare a unităţii de control cablate, permiţându-i cititorului să înţelagă mai uşor mecanismele funcţionale prezentate. În realitate, blocurile RLC, GF şi GIS nu trebuie neaparat separate; ele pot constitui (împreună) o reţea logică secvenţială (RLS) complexă, a cărei proiectare şi funcţionalitate ar fi mult mai dificil de explicat.

2.11 Sistemul de întreruperi şi excepţii

Sistemul de Întrruperi şi Excepţii (SIE) integrat în structura procesorului didactic (fig 2.2) va fi proiectat cu 4 nivele de întreruperi deservite de liniile $IRQ_0 \div IRQ_3$. SIE pune deci la dispoziţie 4 linii de întrerupere externe ($IRQ_0 \div IRQ_3$) pe care se conectează echipamentele periferice (prin intermediul *controller*-elor din blocul I/O). Pe baza acestor 4 linii de intrare pe care sosesc cereri de întrerupere de la periferice, SIE va genera cererea globală de întrerupere INTR care va fi emisă spre BGC şi care va fi testată în punctul interuptibil (la sfârşitul fiecărei instrucţiuni). Pe lângă întreruperi, SIE va fi activat şi de excepţii. Excepţiile pot fi denumite întreruperi interne, deoarece sunt cauzate în general de erori majore detectate în interiorul procesorului şi nu în exterior (nu la nivelul perifericelor). Printre condiţiile care generează excepţii putem aminti:

- cod ilegal (*illegal instruction*); se activează dacă, pe parcursul fazei *instruction fetch* (IF), în registrul IR se încarcă un cod ilegal (cod nealocat).
- împărţire la zero (*divide error*); se activează pe o instrucţiune de împărţire dacă rezultatul obţinut depăşeşte domeniul de reprezentare.
- depăşire stivă (*stack overflow*); se activează pe o instrucţiune *push* care face ca stiva să depăşească zona de memorie alocată.
- tentativă de acces în afara segmentului (*segment fault*); excepţie utilizată pentru implementarea mecanismului de protecţii la nivelul memoriei segmentate.
- acces într-o pagină neprezentă în memoria principală (*page fault*); excepţie utilizată pentru implementarea mecanismului memoriei virtuale.
- căderea tensiunii din reţeaua electrică de alimentare; se semnalează cu ajutorul sursei de alimentare, în momentul în care cade tensiunea alternativă din reţeaua de alimentare (ACLOW).

Tratarea excepţiilor este perfect similară cu cea a întreruperilor. Fiecare excepţie (întrerupere) are asignat propriul vector de excepţie (întrerupere). Excepţiile reprezintă anumite condiţii (erori) detectate în procesor pe parcursul procesării instrucţiunilor. Întreruperile provin din mediul extern (de la periferice). Prin aceste cereri de întrerupere, perifericele solicită procesorului servcii. Serviciile sunt implementate în sistemul de operare (*handler*-ele de întreruperi).

Deoarece perifericele nu sunt sincronizate (nu există semnale de sincronizare între diversele echipamente periferice), rezultă că pot apărea situaţii în care, pe liniile $IRQ_0 \div IRQ_3$, se activează simultan două sau mai multe cereri de întrerupere. Chiar dacă se activează simultan, cererile nu pot fi deservite decât succesiv. Este deci necesară prioritizarea cererilor pentru a stabili ordinea de deservire a acestora. În consecinţă, va fi nevoie de un circuit de reţinere (memorare) a cererilor şi de unul de prioritizare a acestora. Trebuie să mai ţinem cont de faptul că excepţiile sunt mai prioritare decât întreruperile externe şi asemeni întreruperilor, prioritizate între ele. Reamintim faptul că un periferic activeză cereea sa de întrerupere pentru a solicita procesolului un anumit serviciu:

- tastatura activează întreruperea (întreruperea de tastatură) atunci când utilizatorul apasă o tastă, solicitând procesorului să preia codul tastei apăsate. Acest serviciu de preluare a codului este implementat în *handler*-ul de tastatură.

- imprimanta activează întreruperea (întreruperea de imprimantă) atunci când, în procesul de tipărire, goleşte *buffer*-ul de caractere. Prin întrerupere, cere procesorului să încarce în *buffer* următorul şir de caractere (din cadrul fişierului aflat în curs de tipărire). Acest serviciu de încărcare în *buffer* este implementat în *hanler*-ul de imprimantă.

- *mause*-ul activează întrerupera (întreruperea de *mouse*) atunci căND utilizatorul acţionează un buton; prin întrerupere solicită procesorului să execute pe *display* funcţia aferentă butonului acţionat. Acest serviciu este implementat în *hanler*-ul de *mouse*.

Prin urmare, serviciile sunt solicitate prin cereri de întrerupere (semnale activate la nivel *hardware*). La sfârşitul instrucţiunii curente (în punctul interuptibil) procesorul detectează faptul că există o cerere de întrerupere activă şi datorită acestui fapt, va intra în faza de întrerupere (INT). În faza de întrerupere (fig 2.71 şi 2.72), se salvează în stivă registrele FLAG şi PC (starea *flag*-urilor din momentul întreruperii şi adresa de revenire în *task*-ul ce urmează a fi întrerupt) după care, pe baza vectorului de întrerupere, se identifică perifericul care a solicitat întreruperea şi se lansează în execuţie *handler*-ul de întrerupere aferent acesui periferic (un nou *task*). *Handler*-ul va face perifericului serviciul solicitat. După efectuarea serviciului (prin instrucţiunea IRET de la sfârşitul *handler*-ului) se revine în *task*-ul suspendat temporar la activarea cererii de întrerupere. Acesta va fi reluat şi continuat. Prin urmare, perifericele solicită sistemului servicii prin activarea unor semnale la nivel *hardware* (prin activarea cererilor de întrerupere). Suspendarea temporară a *task*-ului curent (a programului aflat în curs de execuţie) si lansarea în execuţie a *handler*-ului de întrerupere (un nou *task* care face parte din sistemul de operare) este realizată de faza de întrerupere (o fază specială implementată la nivel *hardware* şi care nu face parte din ciclul instrucţiunii). Serviciul solicitat de periferic este implementat în *software* (în *handler*-ul de

întrerupere). După efectuarea serviciului solicitat, *task*-ul temporar suspendat va fi reluat şi continuat. În această manieră se elimină timpii de aşteptare la nivelul procesorului (aşteptarea încheierii unor funcţii lansate pe diverse periferice) şi se paralelizează funcţiile lansate pe periferice (operaţiile I/O) cu execuţia programelor în sistem. De exemplu, în timp ce imprimanta tipăreşte şirul de caractere încărcat în *buffer* (ca urmare a activării întreruperii), procesorul continuă execuţia *task*-ului în curs.

2.11.1 Memorarea şi prioritizarea cererilor de întrerupere

Pentru memorarea şi prioritizarea cererilor de întrerupere propunem schema din figura 2.95 în care apar următoarele semnale:

$IRQ_0 \div IRQ_3$ -patru linii de întrerupere (***I**nterrupt **ReQ**uest*) asignate perifericelor; pe fiecare linie se conectează un periferic care, utilizând linia asignată, poate activa o cerere de întrerupere. Semnalele $IRQ_0 \div IRQ_3$ se aplică pe intrările J aferente celor patru bistabili de întrerupere ($BI_0 \div BI_3$). Primul impuls de tact CLK_P care survine după activarea unei cereri IRQ_i, va seta sincron bistabilul aferent (BI_i).

$I_{0p} \div I_{3p}$ -cereri de întrerupere prioritizate, cu I_{0p} cea mai prioritară şi I_{3p} cea mai puţin prioritară. Dacă de exemplu, 3 periferice activează simultan cereri de întrerupere pe IRQ_1, IRQ_2 şi respectiv IRQ_3, acestea vor fi memorate în bistabilii aferenţi (BI_1, BI_2 şi BI_3) şi după traversarea circuitului de prioritizare, se va genera doar $I_{1p}=1$ (cea mai prioritară din setul celor activate). Dacă întreruperile sunt validate, $I_{1p}=1$ va provoca INTR=1 (figura 2.96). La sfârşirul fazei de execuţie aferentă instrucţiunii curente, procesorul va sesiza faptul că cererea globală de întrerupere este activă (INTR=1), va intra în faza de întrerupere INT (figura 2.72), în care cererea INTR va fi achitată (INTA=1). Semnalul global de achitare (INTA=1) se aplică pe intrarea circuitului de validare (cele 4 porti SI din figura 2.95), generând doar $INTA_1=1$, deoarece doar $I_{1p}=1$. Celelalte trei semnale de achitare ($INTA_0$, $INTA_2$ şi $INTA_3$ rămân pe ”0” deoarece sunt mascate de $I_{0p}=I_{2p}=I_{3p}=0$). Prin urmare, doar perifericul a cărui cerere a fost acceptată va primi răspunsul de achitare. Primind achitarea, acest periferic (*controller*-ul I/O aferent acestui periferic) îşi va dezactiva cererea iar SIE va genera vectorul de întrerupere aferent perifericului în cauză şi va încărca acest vector în registrul IVR. Ulterior, pe impulsul I_6 din cadrul fazei INT (figura 2.72), vectorul va fi transferat în registrul ADR iar pe I_7, se va citi din memorie (din tabela IVT) şi va fi încărcată în PC, adresa de start a *handler*-ului de întrerupere. Astfel va fi lansat în execuţie *handler*-ul de întrerupere aferent liniei IRQ_1; *handler*-ul va face serviciul solicitat de perifericul conectat la această linie. Celelalte două întreruperi ramase în aşteptare (I_{2p} şi I_{3p}) vor fi din nou prioritizate, achitate şi servite (tot în ordinea priorităţiilor ce le-au fost asignate).

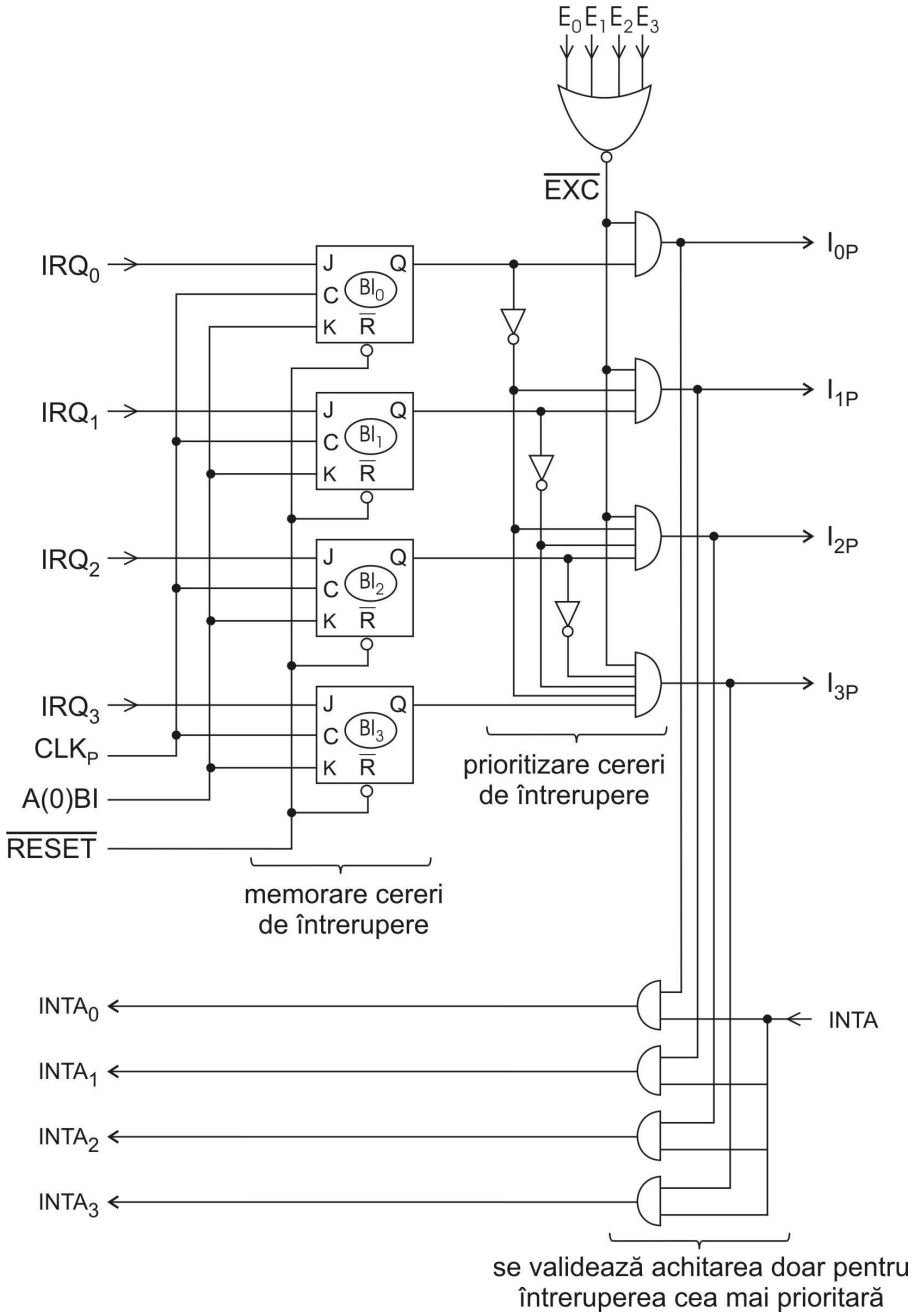


Fig 2.95 Circuitul pentru memorarea şi prioritizarea întreruperilor

În concluzie, chiar şi atunci când mai multe periferice activează simultan cereri de întrerupere (pe liniile IRQ_i), circuitul de prioritizare din figura 2.95 face ca o singură cerere prioritizată (I_{ip}) să fie activă (cea mai prioritară din setul celor activate). Bistabilii BI_i (figura 2.95) sunt resetaţi cu comanda A(0)BI, activată pe impulsul I_7 din cadrul fazei INT (la sfârşitul fazei de întrerupere). Rezultă că cererea prioritizată cea mai prioritară (I_{ip}) este activă pe toată durata fazei INT. Prin urmare, semnalul I_{ip} ar putea fi utilizat în schema de generare a vectorului de întrerupere; fiecărei cereri I_{ip} (i=1÷3) trebuie să i se asocieze propriul vector de întrerupere. Vectorii asociaţi unor linii de întrerupere diferite (periferice diferite), trebuie să fie diferiţi deoarece trebuie să conducă la *handler*-e diferite.

Să observăm (fig 2.95) că la activarea unei excepţii (E_i=1; i=0÷3) se obţine $\overline{EXC}=0$. Semnalul $\overline{EXC}=0$ inhibă toate cererile de întrerupere. Prin urmare, am plecat de la premisa că excepţiile sunt mai prioritare decât întreruperile.

$INTA_0 \div INTA_3$-semnale de achitare destinate perifericelor. Aceste patru semnale de achitare sunt asociate celor patru linii de întrerupere ($IRQ_0 \div IRQ_3$).

Dacă ne referim la exemplul de mai sus, în care se activează simultan cereri de întrerupere pe IRQ_1, IRQ_2 şi IRQ_3 şi în care se va genera I_{1p}=1 (întreruperea cea mai prioritară din setul celor activate), atunci la achitare (când procesorul va activa INTA), din INTA=1 se va obţine $INTA_1$=1 şi $INTA_0$=$INTA_2$=$INTA_3$=0 (figura 2.95). Prin urmare, semnalul de achitare va fi recepţionat doar de perifericul a cărui cerere a fost acceptată. Primind achitarea $INTA_1$=1, perifericul (*controller*-ul I/O) conectat la linia IRQ_1 îşi va dezactiva cererea (IRQ_1, „cade” în ”0”). La sfârşitul fazei INT (pe impulsul I_7), se activează comanda *A(0)BI*. Asta înseamnă că, impulsul de tact (CLK_P) din I_7 va reseta sincron toţi cei patru bistabili de întreruperi $BI_0 \div BI_3$ (figura 2.95). Următorul impuls CLK_P va seta din nou bistabilii BI_2 şi BI_3 deoarece liniile de întrerupere IRQ_2 şi IRQ_3 sunt încă active (doar cererea IRQ_1 a fost anterior achitată şi în consecinţă dezactivată). Prin urmare, cererile IRQ_2 şi IRQ_3 vor fi din nou arbitrate şi deservite succesiv, în ordinea priorităţilor ce le-au fost asignate (mai întâi IRQ_2 şi apoi IRQ_3).

În concluzie, în faza de întrerupere INT (pe primul impuls de stare), BGC activează semnalul global de achitare a întreruperii *INTA* (*INTerrupt Acknowledge*). După cum rezultă din figura 2.95, dacă întreruperea a fost cauzată de cererea I_{iP} (adică dacă I_{iP}=1; i=1÷3), atunci *INTA*=1 va genera:

- $INTA_i=1$; i=1÷3
- $INTA_j$=0; j=1÷3 şi $j \neq i$

Cu alte cuvinte, semnalul global de achitare *INTA* va fi validat de **cererea prioritizată activă** (I_{iP}=1) şi va genera astfel o achitare individuală ($INTA_i=1$) doar pentru perifericul (*cotroller*-ul I/O) care a provocat întreruperea. Celelalte *controller*-e I/O nu vor recepţiona semnalul de achitare ($INTA_j$=0; j=1÷3 şi $j \neq i$).

Să mai observăm că bistabilii de întrerupere $BI_0 \div BI_3$ sunt resetaţi asincron şi de semnalul $\overline{RESET}$, activat atât la acţionarea butonului RESET, cât şi la pornirea calculatorului (din sursa de alimentare). La punerea calculatorului sub tensiune (la pornire), bistabilii $BI_0 \div BI_3$ se poziţionează aleator, pe ”1” sau pe ”0”, datorită hazardului care se manifestă la punerea sistemului sub tensiune. Pentru a nu genera întreruperi false, aceşti bistabili trebuie imediat resetaţi. Această resetare este realizată de semnalul $\overline{RESET}$ care resetează de altfel (la pornire), întreg sistemul de calcul.

2.11.2 Generarea cererii globale de întrerupere INTR

Sistemul de întreruperi şi excepţii trebuie să genereze cererea globală de întrerupere (INTR), care va fi testată de către BGC (unitatea de control) în punctul interuptibil, implementat la sfârşitul fazei de execuţie a fiecărei instrucţiuni. Cererea INTR este deci semnal de ieşire din SIE şi respectiv de intrare în BGC (fig 2.2). Prin urmare, activarea oricărei cereri prioritizate (I_{ip}; i=0÷3) trebuie să conducă în final la activarea cererii globale INTR. Schema circuitului de generare a cereri globale INTR este redată în figura 2.96.

Implementarea din figura 2.96 respectă regula de principiu care stipulează că cererea globală de întrerupere INTR se activează dacă sunt îndeplinite două condiţii (sintetizate în schema din fig. 2.96 pe cele două intrări ale porţii A):

1. cel puţin un periferic solicită întrerupere; transpusă în schema din figura 2.96, această condiţie înseamnă că cel puţin una dintre cererile prioritizate (I_{ip}; i=0÷3) este activă (*high*), ceea ce conduce la $INTR_G$=1

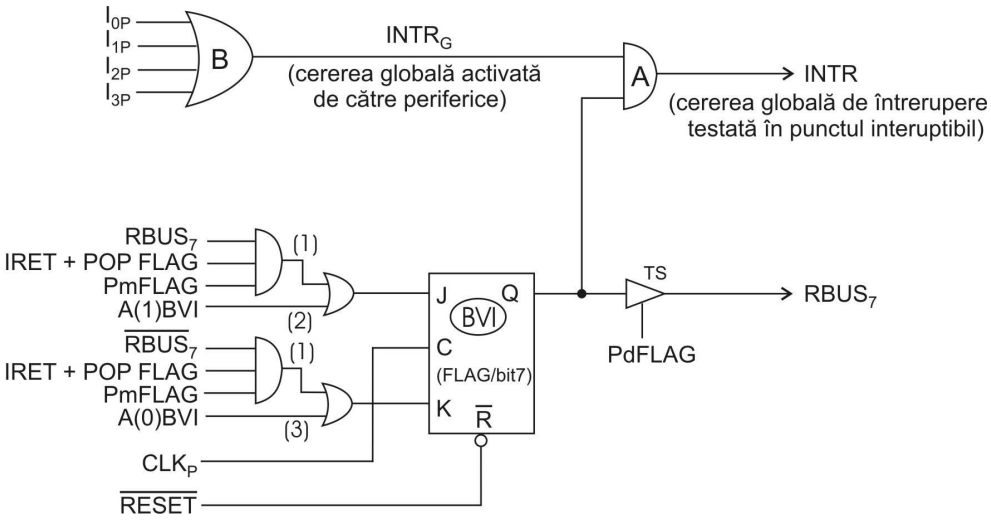
2. întreruperile sunt validate; transpusă în schema din figura 2.96, această condiţie înseamnă că bistabilul de validare a întreruperilor (*flag*-ul BVI din registrul FLAG) este setat.

Dacă cele două condiţii sunt îndeplinite atunci se va obţine INTR=1 pe ieşirea porţii A. Cererea globală INTR este testată în punctul interuptibil (care apare la sfârşitul instrucţiunii în curs de execuţie). Constatându-se INTR=1, se va trece în faza de întrerupere, care va suspenda (temporar) *task*-ul în curs de execuţie şi va lansa în execuţie un nou *task* (*handler*-ul de întrerupere). *Handler*-ul va face serviciul solicitat de către perifericul solicitant.

Validarea/invalidarea întreruperilor se poate realiza cu ajutorul instrucţiunilor:

EI -***E**nable **I**nterrupts* (validare întreruperi)

DI -***D**isable **I**nterrupts* (invalidare întreruperi)



BVI -bistabil de validare a întreruperilor
(bitul 7 din registrul FLAG; fig. 2.27)
(1) -setare/resetare BVI la restaurarea registrului FLAG
din stivă, în cazul instrucţiunilor IRET si POP FLAG
(2) -setare BVI la execuţia instrucţiunii EI
(3) -resetare BVI la execuţia instrucţiunii DI

Fig 2.96 Circuitul de generare a cererii globale de întrerupere INTR

Instrucţiunile EI şi DI vor acţiona asupra bistabilului de validare întreruperi BVI, prin comenzile *A(1)BVI* (setare sincronă bistabil) şi respectiv *A(0)BVI* (resetare sincronă bistabil). Dacă întreruperile sunt invalidate (dacă *flag*-ul BVI este resetat, în urma execuţiei instrucţiunii DI) atunci cererea globala de întrerupere INTR nu poate fi activată (poarta A din fig. 2.96), chiar dacă există una sau mai multe întreruperi prioritizate (I_{ip}; i=0÷3) active (chiar dacă $INTR_G$=1). Întreruperea sau întreruperile (I_{ip}) active vor fi menţinute în aşteptare până la revalidarea întreruperilor (printr-o instrucţiune EI).

BVI este bitul 7 din registrul FLAG. Acesta poate fi deci setat/resetat cu ajutorul instrucţiunilor EI/DI; aceste acţiuni de setare/resetare se produc pe căile notate cu (2)/(3) în figura 2.96.

BVI mai poate fi setat/resetat şi la restaurarea registrului FLAG din stivă (în faza EX a instrucţiunilor IRET şi respectiv POP FLAG). Aceste acţiuni asupra *flag*-ului BVI se produc pe căile marcate cu (1) în schema din fig. 2.96.

Să mai menţionăm că *flag*-ul BVI nu este afectat de instrucţiunea LD FLAG,*src* (nu poate fi setat/resetat cu instrucţiunea LD FLAG,*src* deoarece această instrucţiune s-a introdus în setul de instrucţiuni al procesorului pentru a da posibilitatea programatorului să seteze/reseteze *flag*-urile de condiţii C/Z/S/V şi nu pentru validarea/invalidarea întreruperilor). Cu alte cuvinte, *flag*-ul BVI are un comportament de *flag „read only”* în raport cu instrucţiunea LD FLAG,*src.*

2.11.3 Memorarea şi prioritizarea excepţiilor

Memorarea şi prioritizarea excepţiilor se poate face analog cu cea a întreruperilor externe (fig. 2.97).

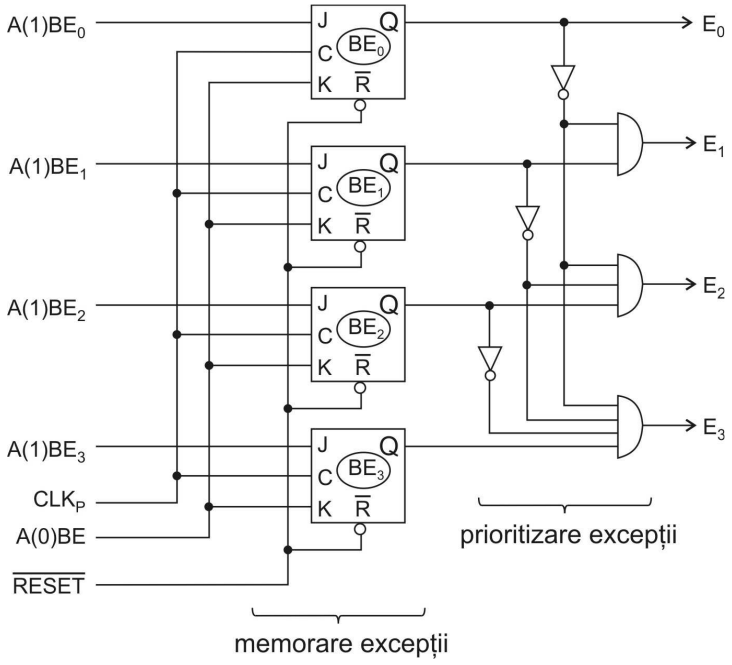


Fig 2.97 Circuitul de memorare şi prioritizare a excepţiilor

Excepţia E_0 (excepţia cea mai prioritară) a fost alocată evenimentului ACLOW (căderea tensiunii alternative din reţeaua de alimentare). Între momentele ACLOW şi DCLOW (căderea tensiunii continue de pe ieşirea sursei de alimentare) apare un interval de timp, datorat energiei înmagazinată în capacităţile de filtrare existente în sursa de alimentare. În acest interval va fi executat *handler*-ul excepţiei E_0 (rutina *POWER FAIL*) care va realiza oprirea de urgenţă a sistemului (nu va fi un *shut down* normal ci unul de urgenţă, în care se salvează pe disc sistemul de operare, exact în starea în care a fost prins de căderea tensiunii de alimentare). La revenirea tensiunii de reţea, se va declanşa rutina *POWER UP* care reîncarcă în memoria principală imaginea salvată anterior pe disc şi redă controlul sistemului de operare exact din punctul în care se afla în momentul activării excepţiei ACLOW.

Excepţia E_1 a fost alocată evenimentului CIL (cod ilegal). Excepţiile E_2 şi E_3 sunt rezervate pentru dezvoltări ulterioare (*stack overflow*, *divide error*, etc.).

Circuitul de prioritizare din figura 2.97 face ca o singură excepţie (E_i) să fie activă (cea mai prioritară din setul celor activate). Bistabilii de memorare a excepţiilor BE_i (figura 2.95) sunt resetaţi cu comanda A(0)BE, activată pe impulsul I_7 din cadrul fazei INT (la

sfârşitul fazei de întrerupere). Rezultă că exceptia cea mai prioritară (E_i) este activă pe toată durata fazei INT. Prin urmare, semnalele E_i pot fi utilizate ca intrări în schema de generare a vectorilor de excepţie (fig 2.99). Fiecărei excepţii E_i (i=1÷3) trebuie să i se asocieze propriul vector de excepţie. Vectorii asociaţi diferitelor excepţii trebuie să fie diferiţi deoarece trebuie să conducă la *handler*-e diferite.

2.11.4 Generarea vectorilor de întrerupere/excepţie

Activarea unei condiţii de excepţie (vezi condiţiile ACLOW şi CIL detectate în faza IF) conduce la setarea bistabilului de excepţie aferent (BE_i) şi intrarea (de urgenţă) în faza INT.

Pe de altă parte, procesarea unei instrucţiuni se încheie întotdeauna cu faza de execuţie iar faza de execuţie se încheie cu punctul interuptibil. Activarea unei întreruperi externe va fi sesizată de către procesor în punctul interuptibil (primul care survine după momentul activării). Sesizată în punctul interuptibil, cererea de întrerupere activată determină intrarea în faza INT.

Prin urmare, sesizarea unei întreruperi (excepţii) active determină procesorul (unitatea de control a acestuia) să suspende (temporar) programul aflat în execuţie şi să lanseze în execuţie un nou program (*handler*-ul de întrerupere/excepţie). Adresele de start aferente *handler*-elor de întrerupere (excepţie) sunt stocate în tabela IVT (*Interrupt Vector Table*). Adresarea locaţiilor tabelei IVT se face cu ajutorul vectorilor de întrerupere (excepţie).

Perifericele sunt conectate (prin intermediul *controller*-elor I/O aferente) la liniile de întrerupere. Pe o linie este conectat un singur periferic. Fiecărei linii de întrerupere (fiecărui periferic) i se alocă propriul vector de întrerupere deoarece fiecare periferic are propriul *handler*. De asemenea, fiecărei excepţii i se alocă propriul vector de excepţie deoarece fiecare excepţie are asociat propriul *handler*. Prin urmare, vectorii de întrerupere (excepţie) trebuie să difere ca valoare. Doi vectori egali (de exemplu cel alocat discului egal cu cel alocat imprimantei) ar selecta în IVT aceiaşi locaţie şi prin urmare ar conduce la acelaşi *handler* (*handler* comun) şi nu la *handler*-e diferite. De asemenea, adresele de start stocate în IVT trebuie să difere ca valoare. Dacă două locaţii ale tabelei IVT conţin aceiaşi adresă de start, această adresă va conduce la un *handler* (*handler* comun) chiar dacă se porneşte de la doi vectori de întrerupere (excepţie) diferiţi.

Sistemul de întreruperi şi excepţii pe care-l proiectăm gestionează patru linii de întrerupere (figura 2.95) şi respectiv patru excepţii (figura 2.97). După memorarea şi prioritizarea cererilor de întrerupere (figura 2.95), se obţin întreruperile prioritizate I_{ip}; (i=0÷3). Chiar şi în cazurile în care mai multe periferice solicită întrerupere simultan, circuitul de prioritizare din figura 2.95 va activa o singură cerere I_{ip} (cea aferentă cererii celei mai prioritare). Similar, după memorarea şi prioritizarea excepţiilor se obţin excepţiile prioritizate E_i; (i=0÷3). Excepţia E_0 a fost alocată evenimentului ACLOW, excepţia E_1 a fost alocată evenimentului CIL iar excepţiile E_2 şi E_3 sunt disponibile pentru dezvoltări; acestea pot fi alocate unor evenimente cum ar fi „*divide error*” (împărţire la zero), „*stack overflow*” (depăşire stivă), etc. Ca în cazul întreruperilor, la un moment dat un singur semnal E_i poate fi activ (cel aferent excepţiei celei mai prioritare). Asta înseamnă că generarea vectorilor de întrerupere (excepţie) poate porni de la semnalele I_{ip} (i=0÷3) şi respectiv E_i (i=0÷3).

Pentru cele patru întreruperi şi respectiv patru excepţii ne propunem să generăm vectorii de întrerupere/excepţie din tabelul 2.10.

Excepţie/întrerupere	Vector de întrerupere/excepţie	
	Valoare în cod hexa	Valoare în cod binar
E_0 (ACLOW)	0000_H	0000000000000000_2
E_1 (CIL)	0002_H	0000000000000010_2
E_2 (nealocată)	0004_H	0000000000000100_2
E_3 (nealocată)	0006_H	0000000000000110_2
IRQ_0 (I_{0p})	0008_H	0000000000001000_2
IRQ_1 (I_{1p})	$000A_H$	0000000000001010_2
IRQ_2 (I_{2p})	$000C_H$	0000000000001100_2
IRQ_3 (I_{3p})	$000D_H$	0000000000001110_2

Tabelul 2.10 Vectorii de întrerupere şi excepţie

Să notăm că, la procesorul didactic, vectorii trebuie să aibă valori pare deoarece adresează în IVT (în memorie) locaţii pe 16 biţi. În aceste locaţii sunt stocate adresele de start ale *handler*-elor de întrerupere/excepţie (adrese pe 16 biţi). Analizând valorile vectorilor de întrerupere/excepţie din tabelul 2.10 putem concluziona faptul că tabela IVT aferentă procesorului didactic conţine 8 locaţii (contigue în memorie) şi este plasată chiar la începutul memoriei principale (fig 2.98).

În tabelul 2.11 sunt explicitaţi cei 16 biţi ai vectorului de întrerupere/excepţie ($V_{15} \div V_0$) şi sunt redate valorile acestor biţi pentru cei opt vectori de excepţie şi respectiv întrerupere definiţi mai sus. Pe baza tabelului 2.11 se pot elabora ecuaţiile (2.15) ce vor sta la baza generării celor 16 biţi ai vectorului de întrerupere/excepţie.

$$
\begin{aligned}
&V_{15}=0\\
&V_{14}=0\\
&\ldots\ldots\ldots\ldots\ldots\ldots\\
&\ldots\ldots\ldots\ldots\ldots\ldots\\
&\ldots\ldots\ldots\ldots\ldots\ldots\\
&V_5=0\\
&V_4=0\\
&V_3=I_{0p}+I_{1p}+I_{2p}+I_{3p}\\
&V_2=E_2+E_3+I_{2p}+I_{3p}\\
&V_1=E_1+E_3+I_{1p}+I_{3p}\\
&V_0=0
\end{aligned}
\qquad (2.15)
$$

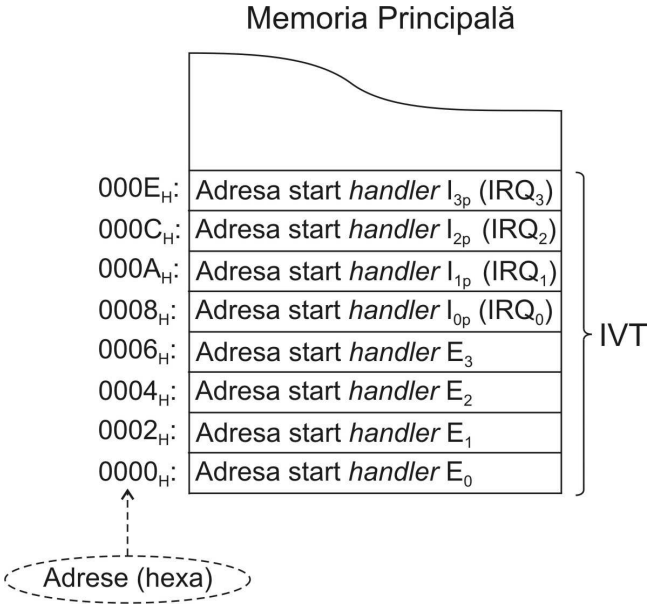


Fig 2.98 Tabela IVT aferentă procesorului didactic

Excepţia (întreruperea) prioritizată	**Valoare vector întrerupere/excepţie (valoare binară)**															
	V_{15}	V_{14}	V_{13}	V_{12}	V_{11}	V_{10}	V_9	V_8	V_7	V_6	V_5	V_4	V_3	V_2	V_1	V_0
E_0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
E_2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
E_3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
I_{0p}	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
I_{1p}	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
I_{2p}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
I_{3p}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

Tabelul 2.11 Valorile binare aferente vectorilor de întrerupere/excepţie

Pe baza ecuaţiilor (2.15) se poate proiecta schema de generare a vectorilor de întrerupere/excepţie (fig 2.99). Semnalele E_i ($i=1\div3$) sunt generate (fig 2.97) prin setarea

bistabililor de memorare a excepţiilor BE_i (i=1÷3). Setarea unui sau mai multor bistabili de excepţie se face în momentul detectării unei sau mai multor condiţii de excepţie. Condiţia va (conditiile vor) determina intrarea procesorului în faza INT. Bistabilul de excepţie revine (bistabilii revin) în starea iniţială cu ajutorul comenzii A(0)BE, activată la sfârşitul fazei INT (pe impulsul I_7). Resetarea bistabilului (bistabililor) de memorare conduce la dezactivarea semnalului E_i (aferent excepţiei celei mai prioritare din setul celor activate). Prin urmare, semnalele E_i sunt active pe toată durata fazei INT. Asta înseamnă că pot fi utilizate ca intrări în schema de generare a vectorului de excepţie (fig 2.99).

În mod analog, semnalele I_{ip} (i=0÷3) sunt active pe toată durata fazei INT (fig 2.95). Prin urmare, pot fi utilizate ca intrări în schema de generare a vectorilor de întrerupere (fig 2.99).

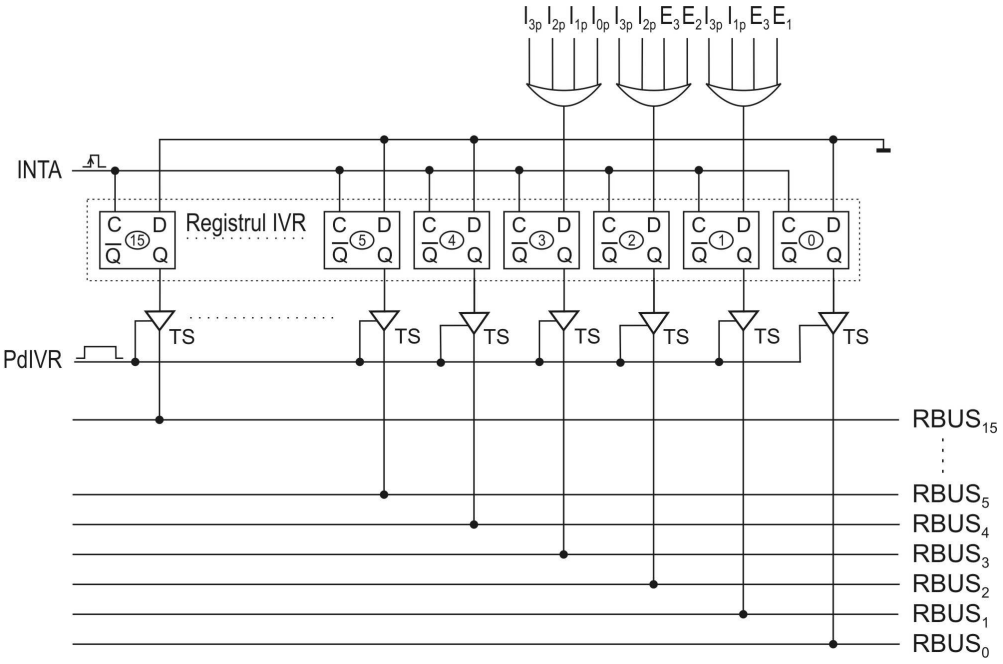


Fig 2.99 Generarea vectorilor de întrerupere/excepţie

Cele 3 porţi SAU din figura 2.99 generează biţii V_3, V_2 şi respectiv V_1 ai vectorului de întrerupere/excepţie; ceilalţi 13 biţi sunt generaţi cu ajutorul unei linii conectate la nivelul logic ”0” ($V_{15}=V_{14}=V_{13}=V_{12}=V_{11}=V_{10}=V_9=V_8=V_7=V_6=V_5=V_4=V_0=0$).

Încărcarea vectorului în registrul IVR se face cu ajutorul comenzii INTA. Prin urmare, încărcarea vectorului în registrul IVR se face în momentul achitării întreruperii. Activarea comenzii INTA (achitarea întreruperii) se realizează pe impulsul I_1 din cadrul fazei INT. Aşadar, pe lângă rolul principal de achitare a întreruperii, comanda INTA are şi rolul de a încărca vectorul de întrerupere în IVR. Prin urmare, perifericul care primeşte achitarea va fi chiar cel care va avea vectorul încărcat în IVR !

Vectorul de întrerupere/excepţie este transferat apoi din registrul IVR în registrul ADR (pentru adresarea tabelei IVT). Acest transfer IVR→ADR este efectuat pe impulsul I_6 din cadrul fazei INT (vezi figurile 2.71 şi 2.72). Pe ultimul impuls de stare din cadrul fazei INT (I_7), se citeşte adresa de start a *handler*-ului din tabela IVT şi se încarcă în registrul PC. Se trece apoi în faza IF unde se va face *fetch*-ul primei instrucţiuni din *handler*. Astfel, este lansat în execuţie *hanler*-ul de întrerupere.

2.11.5 Perfecţionarea sistemului de întreruperi şi excepţii

Sistemul de întreruperi şi excepţii proiectat pe parcursul paragrafelor 2.11.1÷2.11.4 este un sistem simplu care asigură:

- patru linii de întrerupere (***I**nterrupt **ReQ**uest*): $IRQ_0 \div IRQ_3$
- patru excepţii: $E_0 \div E_3$
- prioritizarea întreruperilor şi excepţiilor şi tratarea în ordine priortară a acestora
- validarea/invalidarea (globală) a întreruperilor
- generarea vectorilor de întrerupere/excepţie
- cascadarea întreruperilor (posibilitatea de a întrerupe un *handler* de întrerupere/excepţie; vezi fig 2.10)

Scopul a fost acela de a defini structural un sistem de întreruperi şi excepţii (SIE) simplu, respectând în acest fel paradigma de bază a procesorului didactic: structură simplă, uşor de înţeles.

Este evident că cele patru excepţii şi respectiv cele patru linii de întrerupere sunt insuficiente pentru un procesor comercial. Pentru un astfel de procesor există mai multe condiţii de eroare care induc excepţii (*ACLOW*, *CIL*, *STACK OVERFLOW*, *DIVIDE ERROR*, *SEGMENT FAULT*, *PAGE FAULT*, *DEBUG*, etc) dar numărul acestor condiţii nu depăşeşte 16. Prin urmare, un SIE capabil să gestioneze 16 excepţii ar fi suficient şi pentru un procesor comercial. Dacă dorim astfel de capabilităţi şi pentru procesorul didactic atunci schema din figura 2.97 ar trebui extinsă astfel încât să memoreze şi să prioritizeze 16 excepţii ($E_0 \div E_{15}$). Extensia presupune introducerea unui registru de memorare a excepţiilor pe 16 biţi şi respectiv extinderea circuitului de prioritizare la 15 inversoare şi respectiv 15 porţi SI cu un număr din ce în ce mai mare de intrări (ultima poartă SI va avea 16 intrări). De asemenea, circuitul de generare a vectorilor de întrerupere/excepţie (tabelul 2.11 şi schema din figura 2.99) trebuiesc extinse astfel încât să genereze 16 vectori de excepţie.

Aceeaşi limitare apare şi la nivelul întreruperilor; cele patru linii de întrerupere ($IRQ_0 \div IRQ_3$) sunt insuficiente pentru un procesor comercial. Pe fiecare linie se conectează un singur periferic şi evident că numărul de periferice nu poate fi limitat la patru. Un SIE cu 16 linii de întrerupere ($IRQ_0 \div IRQ_{15}$) şi cu o eventuală posibilitate de extensie (la nevoie) ar fi suficient şi pentru un procesor comercial. Prin urmare, schema din figura 2.95 ar trebui extinsă într-o manieră similară celei din figura 2.97. Vor apărea 16 linii de întrerupere ($IRQ_0 \div IRQ_{15}$), 16 linii de achitare ($INTA_0 \div INTA_{15}$), un registru de memorare a

întreruperilor pe 16 biţi şi un circuit de prioritizare extins care va genera 16 semnale de întrerupere prioritizate ($I_{0P} \div I_{15P}$). De asemenea, circuitul de generare a vectorilor de întrerupere/excepţie (tabelul 2.11 şi schema din figura 2.99) trebuiesc extinse astfel încât să genereze 16 (sau chiar mai mulţi) vectori de întrerupere.

Chiar şi cu aceste extensii, sistemul de întreruperi şi excepţii prezentat până aici nu rezolvă anumite probleme mai rafinate cum ar fi:

- validarea/invalidarea selectivă a întreruperilor (mascarea întreruperilor)
- prioritizarea întreruperilor cascadate

Validarea/invalidarea selectivă sau mascarea întreruperilor se referă la posibilitatea de a valida/invalida, individual, fiecare linie de întrerupere. Asta înseamnă că, pentru un SIE cu 16 linii de întrerupere ($IRQ_0 \div IRQ_{15}$) trebuie introdus un registru de mascare (IMR-*Interrupt Mask Register*) pe 16 biţi. Dacă bistabilul i (bitul i) din registrul de mascare este pe ”1” atunci întreruperile activate pe linia IRQ_i vor fi invalidate (mascate). Dacă bitul în discuţie va fi pe ”0” atunci întreruperile activate pe linia IRQ_i vor fi validate (nemascate). Această validare/invalidare selectivă a întreruperilor se adaugă fireşte la validarea/invalidarea globală, realizată prin instrucţiunile EI (*Enable Interrupts*) şi DI (*Disable Interrupts*), instrucţiuni care acţionează asupra *flag*-ului de validare/invalidare globală (*flag*-ul BVI din figura 2.96).

Prioritizarea întreruperilor cascadate (fig 2.10) se referă la faptul că un *handler* de întrerupere (*handler*-ul curent) n-ar trebui să poată fi (la rândul lui) întrerupt, decât de o întrerupere de prioritate strict mai mare decât cea aferentă întreruperii care a provocat lansarea în execuţie a *handler*-ului curent. Transpunerea acestui principiu pe schema din figura 2.10 conduce la schema de prioritizare a întreruperilor cascadate din figura 2.100, unde am considerat $0 \le h,i,j,k,p \le 15$ (SIE cu 16 linii de întrerupere) şi o schemă de prioritizare care asignează liniei IRQ_0 cea mai înaltă prioritate, respectiv liniei IRQ_{15} cea mai mică prioritate.

Aşa cum rezultă din figura 2.100, *handler*-ul aferent întreruperii IRQ_h nu poate fi întrerupt de întreruperea IRQ_i deoarece $i \ge h$ (IRQ_i are prioritate mai mică sau egală cu IRQ_h) dar poate fi întrerupt de întrerupere IRQ_j deoarece $j<h$ (IRQ_j are prioritate mai mare decât IRQ_h). Acest principiu de prioritizare a întreruperilor cascadate este respectat pe tote nivelele cascadei de întreruperi. Prin urmare, mergând mai departe pe lanţul de întreruperi, *handler*-ul aferent întreruperii IRQ_j nu poate fi întrerupt de întreruperea IRQ_k deoarece $k \ge j$ (IRQ_k are prioritate mai mică sau egală cu IRQ_j) dar poate fi întrerupt de întreruperea IRQ_p deoarece $p<j$ (IRQ_p are prioritate mai mare decât IRQ_j). Asta înseamnă că, dacă SIE dispune de 16 linii de întrerupere ($IRQ_0 \div IRQ_{15}$) atunci lanţul de întreruperi cascadate poate conţine o cascadă de maximum 16 *handler*-e de întrerupere: *handler* IRQ_{15}, urmat de *handler* IRQ_{14}, urmat de *handler* IRQ_{13}, ... , urmat de *handler* IRQ_0.

Pentru implementarea macanismului de prioritizare a întreruperilor cascadate este necesară memorarea priorităţii întreruperii curente (*handler*-ului curent), pe toată durata execuţiei *handler*-ului (din faza INT care lansează în execuţie *handler*-ul şi până la sfârşitul fazei EX aferentă instrucţiunii IRET cu care se încheie *handler*-ul). Memorarea revendică un nou registru; acest registru ar memora prioritatea întreruperii curent servisate şi prin urmare ar putea fi denumit ISR (*In Service Register*). Pentru un SIE cu 16 linii de

întrerupere ($IRQ_0 \div IRQ_{15}$), registrul ISR va avea 16 biţi deoarece, aşa cum am arătat mai sus, lungimea maximă a lanţului de întreruperi (de *hanler*-e) cascadate este 16.

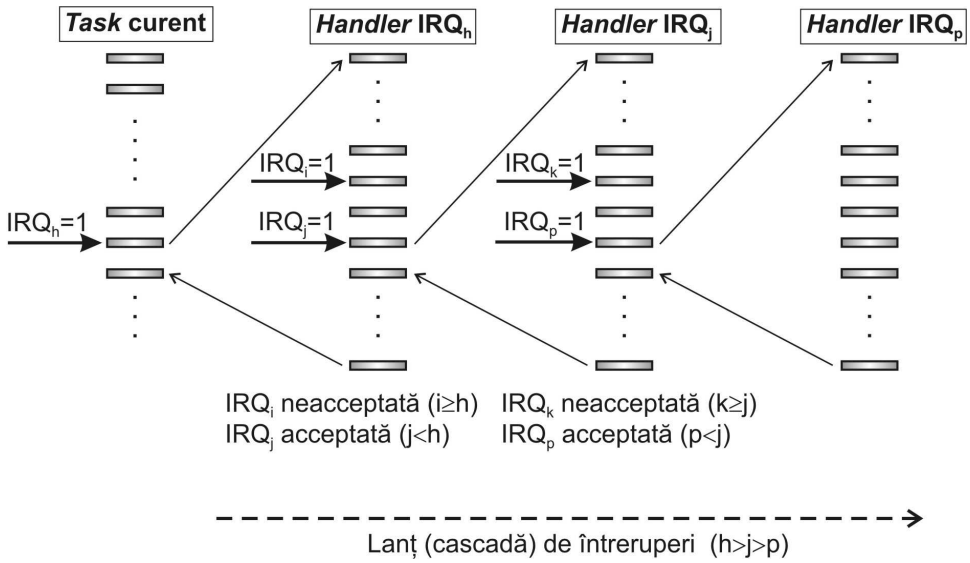


Fig 2.100 Prioritizarea întreruperilor cascadate

În figura 2.101 este redată o schemă de prioritizare care include mascarea întreruperilor, memorarea întreruperilor, prioritizarea întreruperilor simultane, achitarea întreruperilor, şi prioritizarea întreruperilor cascadate. Prin urmare, schema din figura 2.101 soluţionează şi cele două probleme enunţate mai sus ca fiind încă nerezolvate: invalidarea selectivă (mascarea) întreruperilor şi respectiv prioritizarea întreruperilor cascadate. Dat fiind aportul de complexitate indus de noile funcţii (mascarea întreruperilor şi respectiv prioritizarea întreruperilor cascadate), în figura 2.101 este redată doar logica aferentă primelor 4 linii de întrerupere ($IRQ_0 \div IRQ_3$). Pentru un SIE cu 16 linii de întrerupere, schema va fi extinsă (pe verticală) astfel încât să cuprindă şi logica aferentă următoarelor 12 linii de întrerupere ($IRQ_4 \div IRQ_{15}$). Extinderea se va face urmând principiul de extindere (anterior prezentat) aplicat pe schema din figura 2.97. Mai concret spus, vor apărea:

- 16 linii de întrerupere ($IRQ_0 \div IRQ_{15}$)
- 16 linii de achitare ($INTA_0 \div INTA_{15}$)
- registrele IMR, IRR şi ISR extinse la 16 biţi
- circuitele combinaţionale notate cu A, B, C, D, E şi F extinse la nivelul a 16 linii de întrerupere ($IRQ_0 \div IRQ_{15}$) pe intrări, 16 semnale de întrerupere prioritizate ($I_{0P} \div I_{15P}$) pe ieşiri şi respectiv 16 linii de achitare ($INTA_0 \div INTA_{15}$).

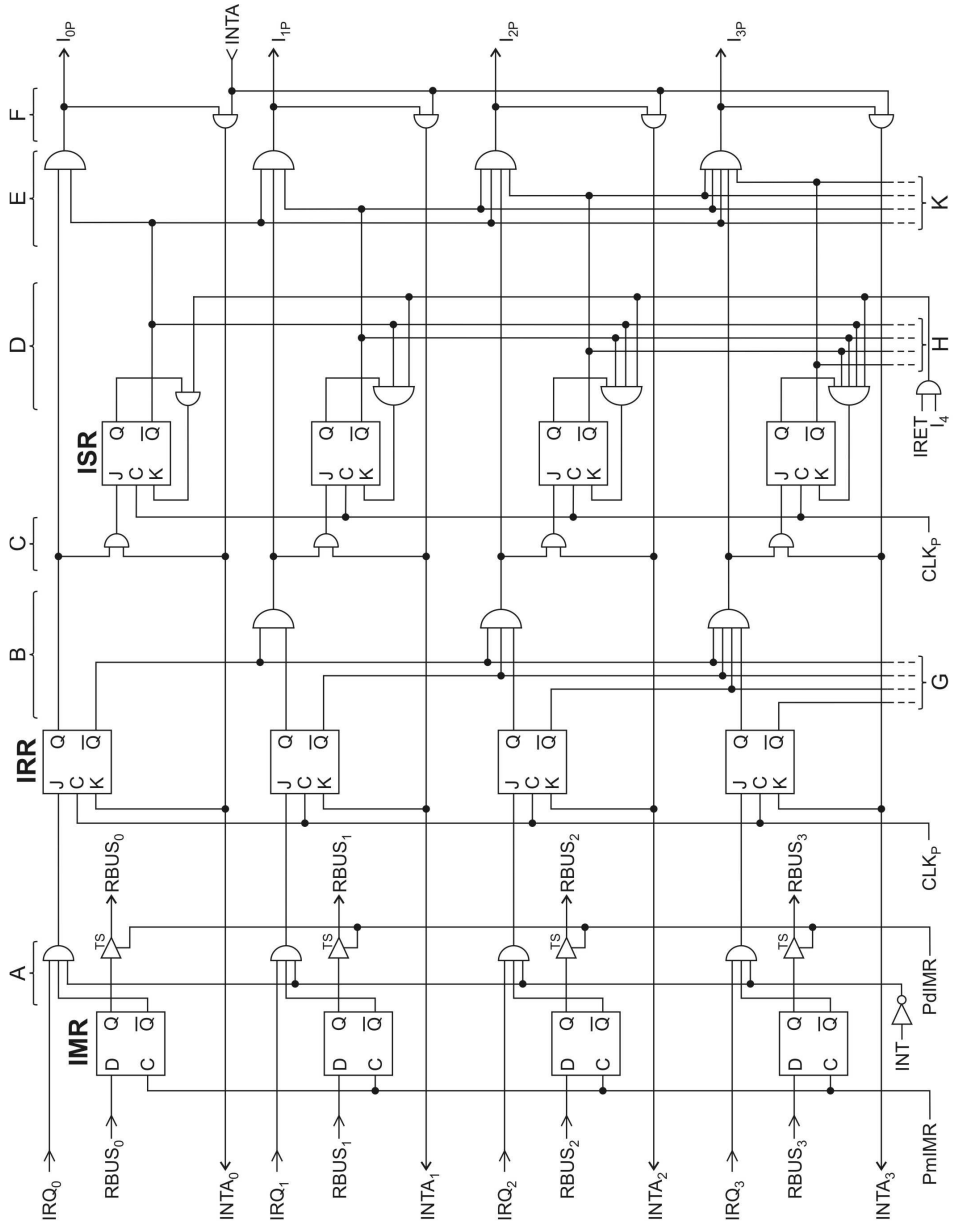


Fig 2.101 Mascarea, memorarea, prioritizarea si achitarea întreruperilor

Din punct de vedere funcţional, în schema din figura 2.101 pot fi identificate următoarele elemente:

- IMR (*Interrupt Mask Register*); în acest registru se încarcă o constantă pe 16 biţi care reprezintă masca (dacă bitul i din mască este pe ”1” atunci întreruperile activate pe linia IRQ_i vor fi invalidate (mascate) iar dacă bitul în discuţie va fi pe ”0” atunci întreruperile activate pe linia IRQ_i vor fi validate (nemascate). Pentru a putea încărca o anumită mască în registrul IMR, acesta trebuie să fie conectat la RBUS. De asemenea, în setul de instrucţiuni aferent procesorului didactic ar trebui adăugată instrucţiunea Ld IMR,*src* (instrucţiune privilegiată) care să permită încărcarea operandului *src* (măştii dorite) în IMR. În această manieră vor putea fi mascate, individual, una sau mai multe linii de întreruperi. Mascarea este realizată cu ajutorul celor 4 (în varianta extinsă 16) porţi SI notate cu A în figura 2.101.

- IRR (*Interrupt Request Register*); în acest registru se memorează starea celor 16 linii de întrerupere ($IRQ_0 \div IRQ_{15}$), amendată cu masca existentă în IMR. Prin urmare, bistabilul i (bitul i) din IRR va fi setat dacă linia IRQ_i este pe ”1” (este activă) şi dacă bitul i din masca stocată în IMR este pe ”0” (dacă IRQ_i este nemascată); dacă bitul i din mască este pe ”1”, atunci bitul i din IRR nu va putea fi setat (chiar dacă IRQ_i=1). Această setare a biţilor din IRR, condiţionată de mască, este realizată cu ajutorul celor 4 (în varianta extinsă 16) porţi SI notate cu A în figura 2.101. Să mai observăm că porţile A sunt validate suplimentar cu INT=0 ceea ce înseamnă că setarea biţilor din IRR se blochează dacă INT=1 (pe durata fazei de întrerupere). Asta e, pe durata fazei INT conţinutul registrului IRR trebuie „îngheţat” deoarece, odată acceptată întreruperea prioritizată I_{iP}=1, starea liniilor I_{iP} trebuie „îngheţată” pe durata fazei INT. Dacă starea liniilor I_{iP} nu ar fi îngeţată pe durata fazei INT atunci riscăm să achităm o întrerupere (cu $INTA_i$=1, la începutul fazei INT) şi să lansăm în execuţie un alt *handler* (mai prioritar), la sfârşitul fazei INT. Să mai observăm că setarea biţilor în IRR se realizează sincron, pe fiecare impuls CLK_P (mai puţin pe impulsurile CLK_P din cadrul fazei INT).

- ISR (*In Service Register*); în acest registru se memorează prioritatea întreruperii curent servisate (prioritatea *handler*-ului curent). Biţii setaţi în IRR (biţi aferenţi cererilor IRQ_i active) sunt prioritizaţi cu ajutorul circuitului de prioritizare (porţile B), rezultând o singură ieşire activă, cea aferentă liniei celei mai prioritare din setul celor active şi nemascate. Prin urmare, dintre cele 16 porţi B, una singură va avea ieşirea activă, cea aferentă liniei celei mai prioritare din setul celor active şi nemascate. Ieşirea activă va provoca setarea bitului i (bistabilului i) din ISR. Setarea se realizează cu validarea $INTA_i$=1 (porţile C), sincron cu tactul procesorului (CLK_P). Prin urmare, setarea se realizează pe impulsul I_1 din cadrul fazei INT, atunci când procesorul achită întreruperea (INTA=1), deoarece din INTA=1 se obţine validarea $INTA_i$=1 (porţile F). Bitul în discuţie rămâne setat în ISR pe toată durata execuţiei *handler*-ului, precizând în acest mod prioritatea întreruperii curent servisate (prioritatea *handler*-ului curent). Bitul va fi resetat doar la sfârşitul *handler*-ului, pe impulsul I_4 din cadrul fazei EX aferentă

instrucţiunii IRET (porţile D). În sfârşit, circuitul de condiţionare implementat cu porţile E nu permite întreruperea unui *handler* (întreruperi cascadate), decât de o întrerupere de prioritate strict mai mare decât cea aferentă *handler*-ului curent (decât cea aferentă întreruperii care a provocat lansarea în execuţie a *handler*-ului curent).

La iniţializarea calculatorului (RESET din buton sau RESET la punerea sub tensiune) se resetează registrele IRR, IMR şi ISR (resetare asincronă, care se realizează aplicând semnalul $\overline{RESET}$ pe intrările $\overline{R}$ aferente bistabililor din componenţa celor trei registre). Pe parcursul funcţionării, atunci când se doreşte mascarea unor întreruperi, sistemul de operare poate seta diverse măşti în IMR (cu instrucţiunea privilegiată Ld IMR,*src*). În timpul funcţionării calculatorului, întreruperile activate sunt luate în considerae în punctul interuptibil, implementat la sfârşitul fiecărei instrucţiuni (la sfârşitul fazei EX).

La activarea unei (sau mai multor) întreruperi, în cadrul schemei din figura 2.101 se vor parcurge următorii paşi:

1. Una (sau mai multe) linii IRQ_i (i=0÷15) se activează. Liniile IRQ_i active şi nemascate (IMR şi porţile A) vor provoca setarea biţilor corespondenţi din IRR. Biţii setaţi în IRR sunt prioritizaţi (porţile B) conducând la I_{iP}=1 (cea mai prioritară din setul întreruperilor setate în IRR).

2. Dacă întreruperile sunt validate (dacă *flag*-ul BVI este setat), I_{iP}=1 conduce la INTR=1 (figura 2.96). În primul punct interuptibil, procesorul sesizează INTR=1 şi drept consecinţă va intra în faza INT.

3. Pe primul tact din faza INT (pe impulsul I_1) procesorul achită cererea INTR (prin activarea semnalului de achitare INTA). Din INTA=1, porţile F vor genera $INTA_i$=1 (achitarea aferentă întreruperii celei mai prioritare din setul celor setate în ISR).

4. Cu $INTA_i$=1, bitul i din ISR (bitul aferent întreruperii I_{iP}, cea mai prioritară din setul celor activate) este setat iar bitul corespondent din IRR este resetat. Asta înseamnă că, întreruperea I_{iP} (provenită din IRQ_i) trece din faza de „*request*" în faza de „*service*". Pentru setarea bitului din ISR operează porţile C şi intrările J aferente celor 16 bistabili din componenţa registrului ISR (setare sincronă) iar pentru resetarea bitului corespondent din IRR sunt utilizate intrările K aferente celor 16 bistabili din componenţa registrului IRR (resetare sincronă).

5. Faza INT continuă. Se va genera vectorul de întrerupere aferent cererii I_{iP} şi pe baza lui se va lansa în execuţie *handler*-ul aferent liniei IRQ_i (aferent perifericului conectat la această linie).

6. La sfârşitul *handler*-ului (pe impulsul I_4 din faza EX aferentă instrucţiunii IRET) bitul i din ISR (cel setat în pasul 4) este resetat. Cu alte cuvinte, terminarea *handler*-ului trebuie să conducă la resetarea bitului corespondent din ISR. Pentru

resetarea bitului i din ISR operează porţile D şi intrările K aferente bistabililor din componenţa registrului ISR (resetare sincronă).

Atâta timp cât procesorul se află în execuţia unui program, altul decât un *handler* de întrerupere, toţi biţii din ISR sunt pe ”0”. După prima întrerupere, pe durata execuţiei *handler*-ului, un singur bit din ISR va fi setat (cel aferent *handler*-ului executat). La revenirea din *handler* (instrucţiunea IRET), bitul aferent din ISR este resetat (din nou toţi biţii din ISR sunt în ”0”). Acest comportament funcţional rămâne valabil atâta timp cât nu avem de-a face cu întreruperi cascadate.

La cascadare, al doilea *handler* (cel care întrerupe primul *handler*; vezi figura 2.100) va conduce la setarea celui de-al doilea bit în ISR. Pe lanţul de întreruperi cascadate, fiecare nou *handler* produce setarea unui nou bit în ISR. De exemplu, când procesorul execută cel de-al cincilea *handler* din lanţ, vom avea cinci biţi setaţi în ISR deoarece cinci *handler*-e au fost lansate în execuţie şi niciunul n-a fost terminat (figura 2.100).

Totuşi, un *handler* nu poate fi întrerupt decât de o întrerupere mai prioritară (un *handler* mai prioritar). Prin urmare, în cazul întreruperilor cascadate, setarea biţilor în ISR nu se poate face decât în sensul creşterii priorităţii (funcţionalitate asigurată de circuitul de condiţionare imlementat cu porţile E, urmat de porţile F şi respectiv C). La revenirea pe lanţul de întreruperi cascadate (la terminarea succesivă a *handler*-elor din lanţ), resetarea biţilor din ISR se va face în ordine înversă (în sensul descreşterii priorităţii). Acest lucru este asigurat de logica de resetare a biţilor din ISR (porţile D). Să mai menţionăm că:

- la setare, întotdeauna se setează un singur bit în ISR (cel aferent *hanler*-ului ce va fi lansat în execuţie); acest lucru este asigurat de circuitul de condiţionare implementat cu porţile E, urmat de porţile F şi respectiv C.

- la resetare, întotdeauna se resetează un singur bit în ISR (cel aferent *handler*-ului care tocmai se încheie); acest lucru este asigurat de logica de resetare implementată cu porţile D.

Microprogramare. Concept şi implementare

3.1 Introducere

Conceptul microprogramării a fost introdus de către Maurice Wilkes la universitatea din Cambridge în anul 1951 (la numai şase ani de la debutul erei calculatoarelor electronice). Conceptul a fost utilizat sporadic până în anul 1964, când IBM lansează sistemul IBM360 a cărei unitate centrală era microprogramată. Unul dintre cele mai importante argumente pentru introducerea microprogramării la sistemul IBM360 a rezultat dintr-un obiectiv strategic al IBM şi anume acela de a putea emula, într-o manieră mai eficientă, vechile sisteme IBM, pe mai noul 360. Prin introduceea aceastei compatibilităţi de jos în sus, IBM urmărea asigurarea utilizatorilor cu privire la trecerea directă a programelor de aplicaţii de pe vechile sisteme pe noul şi mai performantul IBM360. Asigurarea acestei compatibilităţi la nivel *software* între noile şi vechile sisteme a reprezentat o strategie managerială extrem de valoroasă aplicată de IBM.

Tehnica microprogramării se extinde după 1964 şi astăzi este larg utilizată la proiectarea procesoarelor CISC (*Complex Instruction Set Computer*) care sunt dotate de regulă cu unităţi de control microprogramate; spre deosebire de procesoarele RISC (*Reduced Instruction Set Computer*) care sunt de regulă proiectate cu unităţi de control cablate.

Un procesor microprogramat este un procesor dotat cu o unitate de control microprogramată. Această unitate de control microprogramată este ea însăşi un fel de procesor care operează cu microinstrucţiuni. Microinstrucţiunile sunt organizate în microrutine şi acestea sunt stocate („arse”) într-o memorie de tip ROM, numită memorie de control sau memorie de microprogram (MPM – ***M**icro**p**rogram **M**emory*). Execuţia unei instrucţiuni maşină constă în parcurgerea unui lanţ de microrutine (microrutina *fetch* instrucţiune, microrutina *fetch* operand, microrutina de execuţie).

Din punctul de vedere al proiectantului, microprogramarea reprezintă o tehnică deosebit de flexibilă de proiectare a procesoarelor. Prin flexibilitate trebuie să înţelegem faptul că, utilizând aceeaşi platformă *hardware* (aceleaşi registre interne, aceeaşi unitate ALU, aceleaşi busuri interne etc.) dar cu microprograme diferite, este posibil să emulăm procesoare diferite; cu alte cuvinte, să emulăm maşini virtuale diferite. Prin urmare, un procesor microprogramat este un procesor primar capabil să emuleze o diversitate de procesoare virtuale, situate pe un nivel ierarhic superior celui primar. Memoria de microprogram conţine o colecţie de microrutine (microrutină *fetch* instrucţiune, microrutine *fetch* operand, microrutine de execuţie, microrutină de întrerupere) şi această colecţie de

microrutine se numeşte microprogram de emulare, deoarece emulează setul de instrucţiuni aferent procesorului virtual. Asta inseamnă că, modificând colecţia de microrutine (modificând doar conţinutul memoriei de microprogram, nu şi platforma *hardware* pe care operează acest microcod), se pot emula procesoare diferite. Un procesor emulat este în fond un procesor virtual. Utilizatorul (programatorul) vede doar procesorul virtual (nu şi pe cel primar care îl emulează).

Nota Bene: Programatorul poate „coborâ" cel mult până la nivelul setului de instrucţiuni maşină. În cazul microprogramării acest set de instrucţiuni este emulat în microcod (utilizând colecţia de microrutine). Prin urmare, este setul de instrucţiuni al unui procesor virtual (o măşină virtuală emulată în microcod de un aşa zis procesor primar).

3.2 Schema bloc de principiu a unităţii de control microprogramate

În mod similar cu unitatea de control cablată (figura 2.83), unitatea de control microprogramată va genera comezile elementare care vor conduce la efectuarea operaţiilor elementare dorite. Schema bloc de principiu a unităţi de control microprogramate este redată în figura 3.1.

Asemeni unităţii de control cablate, unitatea microprogramată poate fi privită ca un automat secvenţial complex care funcţionează pe baza semnalului de tact procesor CLK_P obţinut pe ieşirea generatorului de tact (GT). Din figura 3.1 observăm că de fapt CLK_P este semnalul de tact cu care funcţionează automatul SEQ (***Seq****uencer*). Automatul SEQ este un automat simplu cablat care generează comenzile interne, comenzile interne fiind comenzile care controlează operaţiile din interiorul unităţii de control microprogramate.

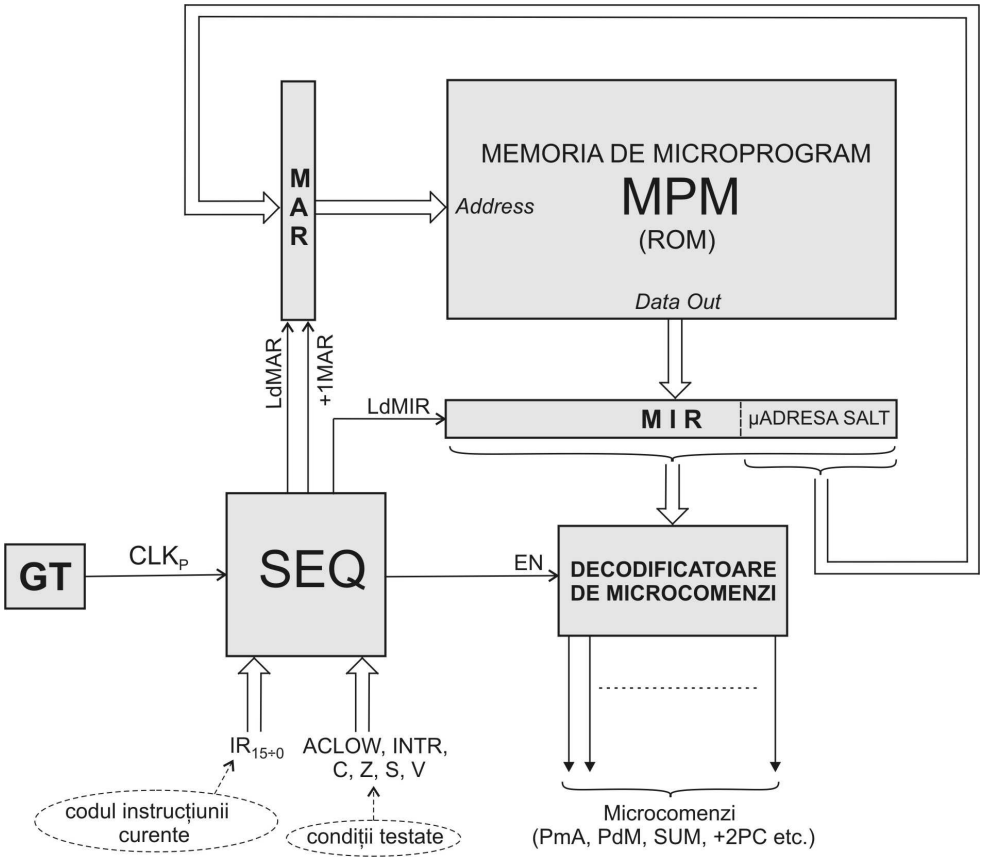
Procesorul proiectat va fi un procesor emulat. Setul de instrucţiuni al procesorului emulat (al procesorului virtual) se defineşte de către proiectant. În acest capitol vom proiecta unitatea de control microprogramată aferentă procesorului didactic. Prin urmare, pentru noi, procesorul virtual va fi procesorul didactic cu arhitectura şi respectiv setul de instrucţiuni stabilite în capitolul 2.

Pentru fiecare instrucţiune a procesorului virtual se „arde" în MPM un microprogram format dintr-o succesiune de microrutine (microrutină ***fetch*** **instrucţiune**, microrutină ***fetch*** **operand**, microrutină de **execuţie**). După cum se observă, fazele definite la unitatea de control cablată devin microrutine la unitatea de control microprogramată. O microrutină este formată dintr-o succesiune de microinstrucţiuni. Aceste microinstrucţiuni sunt procesate secvenţial, una câte una. SEQ încarcă în MIR succesiv aceste microinstrucţiuni. După încărcarea unei microinstrucţiuni în MIR, din biţii acesteia (de regulă prin decodificarea diverselor câmpuri din microinstrucţiune) se obţine un set de comenzi elementare (C_L). Acest set de comenzi elementare, care la unitatea de cntrol microprogramată se numesc de regulă **microcomenzi**, declanşează una sau mai multe operaţii elementare concurente. În cazul unităţii de control cablate, un set de comenzi elementare (concurente) era codificat pe o anumită coloană din cadrul unui tabel de comenzi (pe un impuls de stare). În varianta microprogramată, acest set de comenzi elementare este codificat într-o microinstrucţiune. Prin execuţia, microinstrucţiune cu microinstrucţiune, a microprogramului alocat în MPM unei anumite instrucţiuni, se va obţine acea succesuiune de comenzi (operaţii) elementare care va conduce în final la execuţia instrucţiunii respective.

O microinstrucţiune este procesată principial în doi paşi (două perioade de tact succesive):

1. *fetch*-ul microinstrucţiunii (microinstrucţiunea selectată în MPM de microadresa din MAR este citită şi încărcată în MIR). Pentru încărcarea în MIR, SEQ activează comanda *LdMIR* (*Load* MIR).

2. execuţia microinstrucţiunii (decodificarea microinstrucţiunii din MIR şi activarea comenzilor C_L (microcomenzilor) codificate în microinstrucţiune). Pentru activarea microcomenzilor codificate în microinstrucţiune, SEQ validează, prin comanda *EN* („*Enable*"), decodificatoarele de microcomenzi.



MPM –***M**icro**p**rogram **M**emory* (memoria de microprogram)
MAR –***M**icro**a**ddress **R**egister* (registrul de microadrese)
MIR –***M**icro**i**nstruction **R**egister* (registrul microinstrucţiunii)
SEQ –***Seq**uencer* (automatul secvenţiator)

Fig 3.1 Structura unităţii de control microprogramate.

Tot în pasul doi, în paralel cu execuţia microinstrucţiunii curente, se încarcă în MAR microadresa următoarei microinstrucţiuni. Pentru încărcarea acestei microadrese în MAR,

SEQ activează comanda +1*MAR* sau *LdMAR*. În interiorul unei microrutine, SEQ trece secvenţial de la microinstrucţiunea curentă la cea următoare (activând comanda +1*MAR*). De pe sfârşitul unei microrutine, SEQ face salt pe începutul microrutinei următoare cu ajutorul comenzii *LdMAR* care încarcă în MAR un câmp din microinstrucţiunea curentă (un câmp din MIR). În acest câmp se codifică microadresa de salt.

Pentru procesarea unei microinstrucţiuni sunt necesari deci doi paşi succesivi (două perioade de tact succesive). Pe de altă parte, în microinstrucţiune este codificat un set de comenzi care echivalează, în varianta cablată, cu setul aflat pe o anumită coloană a unui tabel de comenzi (set executat într-o singură perioadă de tact). Rezultă de aici că procesorul microprogramat ar fi de două ori mai lent decât echivalentul său cablat. Întra-devăr microprogramarea este o soluţie mai lentă decât cablarea, dar nu de două ori mai lentă deoarece SEQ-ul este un automat foarte simplu care controlează unitatea de control microprogramată (controlează o structură *hardware* simplă) şi prin urmare poate accepta un tact de frecvenţă mai mare.

Principalul avantaj al microprogramării este flexibilitatea. Procesorul emulat (setul de instrucţiuni emulat) este definit practic de conţinutul MPM (microprogramul de emulare). Rezultă că procesorul virtual (setul de instrucţiuni al acestuia) poate fi complet schimbat prin simple modificări în conţinutul MPM. Deci ajustarea setului de instrucţiuni (modificarea unor instrucţiuni sau adăugarea unor instrucţiuni noi) poate fi făcută chiar şi la finalul fazei de proiectare, prin simple modificări în conţinutul MPM (fără modificări în *hardware*). În varianta cablată, ajustarea setului de instrucţiuni nu este posibilă decât cu modificări în *hardware* (se modifică sau se introduc noi tabele de comenzi ceea ce conduce la modificarea sistemului de ecuaţii logice de la 2.10.3 şi deci la modificări în structura reţelei RLC).

Al doilea avantaj al microprogramării îl reprezintă faptul că proiectarea unităţii de control este mai puţin laborioasă şi mai bine structurată. Unitatea de control microprogramată are o structură simplă, de procesor care operează cu microinstrucţiuni. În susţinerea acestei ultime afirmaţii evidenţiem următoarele similitudini structurale:

- memoria MPM de la nivelul microinstrucţiunii este echivalenta memoriei principale de la nivelul instrucţiunii

- în faza de *fetch* microinstrucţiune, microinstrucţiunea extrasă din MPM este încărcată în MIR. Rezultă că registrul MIR (de la nivelul microinstrucţiunii) este echivalentul registrului IR (de la nivelul instrucţiunii)

- registrul MAR adresează microinstrucţiunile în MPM. MAR joacă deci rol de registru pointer pentru microinstrucţiuni, fiind echivalentul registrului PC de la nivelul instrucţiunii.

3.3 Fazele proiectării unui procesor microprogramat

Proiectarea unui procesor micropragramat curpinde două faze succesive:

1. Faza *hardware*

2. Faza *firmware*

Faza *hardware* cuprinde următoarele etape:

1.a) stabilirea arhitecturii procesorului (schema bloc a procesorului şi setul de instrucţiuni al acestuia)

1.b) stabilirea arhitecturii unităţii de control microprogramate (schema bloc a unităţii de control şi formatul microinstrucţiunii)

1.c) proiectarea *hardware* a tuturor blocurilor din componenţa unităţii de control microprogramate (MPM, SEQ, MIR, MAR, decodificatoare de microcomenzi).

1.d) proiectarea *hardware* a tuturor blocurilor din componenţa procesorului (ALU, registre, busuri etc.)

Faza *firmware* constă în proiectarea microprogramului de emulare aferent setului de instrucţiuni definit la punctul 1.a) de la faza *hardware*. Cu alte cuvinte în faza *firmware* trebuie proiectat conţinutul memoriei de microprogram (MPM).

Obiectivul acestui capitol este proiectarea unui procesor microprogramat. Pentru realizarea acestui obiectiv trebuie să parcurgem fazele (*hardware* şi *firmware*) şi respectiv etapele definite în cadrul acestor faze. Pentru a reduce volumul de muncă pe care o astfel de proiectare îl implică, vom lua o decizie care ne va permite să reutilizăm părţi consistente din capitolul 2: procesorul microprogramat va fi procesorul didactic introdus în capitolul 2. Asta înseamnă că etapele 1.a) şi respectiv 1.d) din faza *hardware* au fost deja parcurse (in capitolul 2) deoarece:

- arhitectura procesorului microprogramat este cea stabilita în capitolul 2 cu schema bloc din figura 2.2 şi cu setul de instrucţiuni definit la 2.2.

- structura *hardware* aferentă blocurilor din componenţa procesorului este cea stabilită pe parcursul capitolului 2 (cu excepţia unităţii de control care nu va mai fi cablată ci microprogramată).

De asemenea, primul pas din etapa 1.b) îl considerăm, la nivel principial, parcurs (schema bloc de principiu aferentă unităţii de control microprogramate este cea din figura 3.1). Evident că această schemă bloc va fi detaliată dar detalierea se va putea realiza doar după proiectarea formatului microinstrucţiunii, adică după parcurgerea pasul doi din cadrul etapei 1.b).

3.4 Formatul microinstrucţiunii

Pentru unitatea de control microprogramată aferentă procesorului didactic vom defini trei tipuri de microinstrucţiuni:

- microinstrucţiunea de ramificaţie

- microinstrucţiunea de transfer sincron

- microinstrucţiunea de transfer asincron

În cele ce urmează, vom stabili formatul aferent fiecărui tip şi vom clarifica operaţiile realizabile cu fiecare tip de microinstrucţiune.

3.4.1 Formatul microinstrucţiunii de ramificaţie

Microprogramul de emulare cuprinde următoarea colecţie de microrutine:

- microrutină *fetch* instrucţiune
- microrutine *fetch* operand (fiecare mod de adresare va avea propria microrutină *fetch* operand)
- microrutine de execuţie (fiecare instrucţiune va avea propria microrutină de execuţie)
- microrutină de întrerupere

În cazul unui procesor microprogramat, procesarea unei instrucţiuni constă în parcurgerea unui lanţ de microrutine care începe întotdeauna cu microrutina *fetch* instrucţiune şi se încheie cu microrutina de execuţie. Prin urmare, fazele (*fetch* instrucţiune, *fetch* operand, execuţie şi întrerupere) din cazul procesorului cablat se transformă în microrutine în cazul procesorului microprogramat.

O microrutină este o secvenţă de microinstrucţiuni parcurse secvenţial. Cum procesarea unei instrucţiuni înseamnă parcurgerea unui lanţ de microrutine, înseamnă că, pe lângă proiectarea microrutinelor, trebuie implementat şi un mecanism eficient care să realizeze legăturile între microrutine.

Microinstrucţiunile de ramificaţie sunt de fapt microinstrucţiuni de salt (condiţionat şi necondiţionat). Apar la sfârşitul fiecărei microrutine pentru a stabili legătura spre microrutina următoare (fac salt la microrutina următoare). Prin urmare, cu ajutorul microinstrucţiunilor de ramificaţie vom implementa legăturile între microrutine.

Formatul microinstrucţiunii de ramificaţie precizează semnificaţia diverselor câmpuri şi respectiv biţi din codul microinstrucţiunii. Acest format este redat în figura 3.2.

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Selecţie condiţie de ramificaţie *f* (SUCCESOR)				Selecţie INDEX		$\overline{T}/F$	MICROADRESA DE SALT											

Fig 3.2 Formatul microinstrucţiunii de ramificaţie

Bitul MIR_{19} specifică tipul microinstrucţiuni:

MIR_{19}=0 –defineşte microinstrucţiunea de ramificaţie

MIR_{19}=1 –defineşte microinstrucţiunea de transfer (de transfer sincron sau asincron)

Bitul MIR_{12} (bitul $\overline{T}/F$) specifică modul de test a condiţiei de ramificaţie *(f)*:

MIR_{12}=0 –specifică *TRUE* (saltul se face pe condiţie adevărată, adică dacă *f*=1)

MIR_{12}=1 –specifică *FALSE* (saltul se face pe condiţie falsă, adică dacă $\overline{f}$ =1)

Funcţia *f* (codificată în câmpul $MIR_{18\div15}$) se numeşte generic <u>condiţie de ramificaţie</u>. Prin urmare, saltul (în microcod) se va efectua numai dacă condiţia de ramificaţie (*f*), testată pe adevărat sau pe fals (în funcţie de valoarea bitului MIR_{12}), este îndeplinită.

La unitatea de control cablată, în tabele de comenzi aferente diverselor faze şi respectiv instrucţiuni, au fost testate o serie de condiţii. Să enumerăm câteva dintre condiţiile testate:

- ACLOW, CIL şi IOP –în tabelul de comenzi aferent fazei IF
- AM, AD, AI şi AX –în tabelul de comenzi aferent fazei OF
- C, Z, S, V –în fazele de execuţie aferente instrucţiunilor de salt condiţionat.
- INTR –în punctul interuptibil din fazele de execuţie aferente tuturor instrucţiunilor (cu excepţia instrucţiunilor HALT şi DI).

Tabelele de comenzi (aferente diverselor faze) elaborate la proiectarea unităţii de control cablate (pe parcursul capitolului 2), devin microrutine la unitatea de control microprogramată. Prin urmare, condiţiile testate în diversele tabele de comenzi vor fi testate şi în cadrul microrutinelor echivalente. Drept urmare, condiţia (generică) de ramificatie *(f)* va trebui generată pornind de la aceste condiţii. Condiţia de testat este selectată de câmpul $MIR_{18\div15}$. Altfel spus, câmpul $MIR_{18\div15}$ trebuie să asigure:

- *f*=C, atunci când se doreşte testarea *flag*-ului *Carry*

- *f*=CIL, atunci când se doreşte testarea condiţiei „cod ilegal”, adică dacă instrucţiunea din IR este instrucţiune ilegală

.....

- *f*=INTR, atunci când se doreşte testarea cererii globale de întrerupere

Pe de altă parte, trebuie introdusă posibilitatea testării, fie pe adevărat fie pe fals, a tuturor condiţiilor. Pentru aceasta, se introduce <u>funcţia globală de ramificaţie</u> *(g)*, care se implementează în acord cu ecuaţia (3.1).

$$g=f\oplus MIR_{12} \qquad (3.1)$$

În final, pentru a decide dacă condiţia testată este îndeplinită (dacă saltul se va face sau nu), este suficientă testarea funcţiei *g*:

- dacă *g*=1 înseamnă că avem **condiţie îndeplinită**, oricare ar fi condiţia testată şi oricare ar fi modul de test (pe fals sau pe adevărat).

- dacă g=0 înseamnă că avem **condiţie neîndeplinită**, oricare ar fi condiţia testată şi oricare ar fi modul de test (pe fals sau pe adevărat).

Prin urmare, saltul se va face numai dacă funcţia globală de ramificaţie (g) este egală cu 1. Dacă $g=0$ saltul devine inefectiv (se execută succesorul *STEP* prin care se trece secvenţial la următoarea microinstrucţiune). Evident că funcţia g trebuie implementată în *hardware*.

Câmpul de selecţie a condiţiei de ramificaţie f (câmpul $MIR_{18\div15}$) împreună cu bitul MIR_{12} are rolul de a decide dacă saltul se va efectua sau nu. Prin urmare $MIR_{18\div15}$, împreună cu bitul MIR_{12}, decid care va fi succesoarea microinstrucţiunii curente: microinstrucţiunea de la MICROADRESA DE SALT (codificată în câmpul $MIR_{11\div0}$) sau cea de la microadresa secvenţial următoare. Din acest motiv câmpul $MIR_{18\div15}$ se mai numeşte şi SUCCESOR (determină succesoarea microinstrucţiunii curente).

Câmpul $MIR_{11\div0}$ (câmpul MICROADRESA DE SALT) furnizează microicroadresa pe care se va executa saltul (atunci când g=1). Această microadresă este interpretată însă ca o microadresă de bază la care se adună un index. Indexul este selectat de câmpul $MIR_{14,13}$. Dacă $MIR_{14,13}$=0,0 atunci indexul selectat va avea valoarea zero (vezi tabelul 3.3). În acest caz saltul se va face chiar la microadresa de bază preluată din câmpul $MIR_{11\div0}$ al microinstrucţiunii curente. Succesorii care vor fi obţinuţi în acest caz vor fi succesori JUMP (salturi simple, neindexate). Succesorii JUMP sunt prezentaţi în tabelul 3.1.

$MIR_{18\div15}$	Mnemonica microinstrucţiunii	Funcţia f testată
0000	JUMP μADR (μADR=$MIR_{11\div0}$)	$f=\overline{MIR}_{12}$, (g=1), salt necondiţionat
0001	IF (N)IOP JUMP μADR else STEP	f=IOP=$\overline{IR}_{15}$ (clasa de instrucţiuni IOP)
0010	IF (N)AM JUMP μADR else STEP	$f=\overline{IR}_9 \cdot \overline{IR}_8$ (adresare imediată)
0011	IF (N)AD JUMP μADR else STEP	$f=\overline{IR}_9 \cdot IR_8$ (adresare directă)
0100	IF (N)AI JUMP μADR else STEP	$f=IR_9 \cdot \overline{IR}_8$ (adresare indirectă)
0101	IF (N)AX JUMP μADR else STEP	$f=IR_9 \cdot IR_8$ (adresare indexată)
0110	IF (N)INTR JUMP μADR else STEP	f=INTR (cererea globală de întrerupere)
0111	IF (N)ACLOW JUMP μADR else STEP	f=ACLOW (căderea tensiunii de alimentare)
1000	IF (N)CIL JUMP μADR else STEP	f= CIL (cod ilegal)
1001	IF (N)C JUMP μADR else STEP	f=C (*flag*-ul *Carry* din registrul FLAG)
1010	IF (N)Z JUMP μADR else STEP	f=Z (*flag*-ul *Zero* din registrul FLAG)
1011	IF (N)S JUMP μADR else STEP	f=S (*flag*-ul *Sign* din registrul FLAG)
1100	IF (N)V JUMP μADR else STEP	f=V (*flag*-ul *Overflow* din registrul FLAG)
1101	Neutilizată (rezervată pentru dezvoltări)	-
1110	Neutilizată (rezervată pentru dezvoltări)	-
1111	Neutilizată (rezervată pentru dezvoltări)	-

Tabelul 3.1 Succesorii JUMP (codificaţi în câmpul $MIR_{18\div15}$), dacă $MIR_{14,13}$=0,0)

Dacă $MIR_{14,13} \neq 0,0$ atunci se va selecta un index diferit de zero (vezi tabelul 3.3). În acest caz saltul se va face la MICROADRESA DE SALT+INDEX (microadresa de bază + indexul selectat). Precizăm din nou că microadresa de bază este codificată în câmpul $MIR_{11 \div 0}$ iar indexul este selectat cu câmpul $MIR_{14,13}$. Succesorii devin în acest caz JUMPI (salturi indexate). Succesorii JUMPI sunt prezentaţi în tabelul 3.2.

MIR $_{18 \div 15}$	Mnemonica microinstrucţiunii	Funcţia *f* testată
0 0 0 0	JUMPI μADR+INDEX (μADR=$MIR_{11 \div 0}$)	$f = \overline{MIR}_{12}$, (g=1), salt necondiţionat
0 0 0 1	IF (N)IOP JUMPI μADR+INDEX else STEP	f=IOP= $\overline{IR}_{15}$ (clasa de instrucţiuni IOP)
0 0 1 0	IF (N)AM JUMPI μADR+INDEX else STEP	$f = \overline{IR}_9 \cdot \overline{IR}_8$ (adresare imediată)
0 0 1 1	IF (N)AD JUMPI μADR+INDEX else STEP	$f = \overline{IR}_9 \cdot IR_8$ (adresare directă)
0 1 0 0	IF (N)AI JUMPI μADR+INDEX else STEP	$f = IR_9 \cdot \overline{IR}_8$ (adresare indirectă)
0 1 0 1	IF (N)AX JUMPI μADR+INDEX else STEP	$f = IR_9 \cdot IR_8$ (adresare indexată)
0 1 1 0	IF (N)INTR JUMPI μADR+INDEX else STEP	f=INTR (cererea globală de întrerupere)
0 1 1 1	IF (N)ACLOW JUMPI μADR+INDEX else STEP	f=ACLOW (căderea tensiunii de alimentare)
1 0 0 0	IF (N)CIL JUMPI μADR+INDEX else STEP	f=CIL (cod ilegal)
1 0 0 1	IF (N)C JUMPI μADR+INDEX else STEP	f=C (*flag*-ul *Carry* din registrul FLAG)
1 0 1 0	IF (N)Z JUMPI μADR+INDEX else STEP	f=Z (*flag*-ul *Zero* din registrul FLAG)
1 0 1 1	IF (N)S JUMPI μADR+INDEX else STEP	f=S (*flag*-ul *Sign* din registrul FLAG)
1 1 0 0	IF (N)V JUMPI μADR+INDEX else STEP	f=V (*flag*-ul *Overflow* din registrul FLAG)
1 1 0 1	Neutilizată (rezervată pentru dezvoltări)	-
1 1 1 0	Neutilizată (rezervată pentru dezvoltări)	-
1 1 1 1	Neutilizată (rezervată pentru dezvoltări)	-

Tabelul 3.2 Succesorii JUMPI (codificaţi în câmpul $MIR_{18 \div 15}$), dacă $MIR_{14,13} \neq 0,0$)

Câmpul $MIR_{14,13}$ are deci rol de selector de index. Cu 2 biţi pot fi selectaţi 4 indecşi dintre care, după cum am afirmat mai sus, primul va fi zero. Indecşii definiţi sunt prezentaţi în tabelul 3.3.

Din tabelul 3.3 se observă că am definit valorile binare aferente primilor 3 indecşi ($INDEX_1$, $INDEX_2$ şi $INDEX_3$). Pentru că sunt necesari doar trei indecşi, ultima combinaţie ($MIR_{14,13}$=1,1) este rezervată pentru dezvoltări viitoare. Prin urmare, $INDEX_4$ nu a fost definit ca valoare.

Indexul selectat se adună la microadresa de bază pentru a genera microadresa pe care se va executa saltul. Din tabelul 3.3 se observă ca, în afara primului index ($INDEX_1$) care are valoarea zero şi care are rolul de a transforma saltul indexat într-unul neindexat

(transformă succesorii JUMPI din tabelul 3.2 în succesorii JUMP corespondenţi din tabelul 3.1), ceilalţi 2 indecşi ($INDEX_2$ şi $INDEX_3$) preiau biţii câmpului OPCODE şi respectiv MOD DE ADRESARE din instrucţiunea curentă.

MIR_{14}	MIR_{13}	Denumire INDEX	Valoare INDEX (valoare binară)											
0	0	$INDEX_1$	0	0	0	0	0	0	0	0	0	0	0	0
0	1	$INDEX_2$	0	0	0	0	0	IR_{14}	IR_{13}	IR_{12}	IR_{11}	IR_{10}	0	0
1	0	$INDEX_3$	0	0	0	0	0	0	0	0	IR_9	IR_8	0	0
1	1	$INDEX_4$	(combinaţie neutilizată; disponibilă pentru dezvoltări viitoare)											

Tabelul 3.3 Indecşii selectaţi de câmpul $MIR_{14,13}$ din microinstrucţiune

Observaţie:
Dacă INDEX-ul are valoarea zero atunci succesorii JUMPI µADR+INDEX din tabelul 3.2 devin:

JUMPI µADR+INDEX=JUMPI µADR+0=JUMP µADR

Prin urmare, tabelul 3.1 reprezintă un caz particular al tabelului 3.2.

A. Successorii JUMP µADR

Successorii JUMP µADR sunt prezentaţi în tabelul 3.1. Prima microinstrucţiune din tabelul 3.1 este un salt simplu necondiţionat, iar următoarele 12 sunt salturi simple condiţionate. Execuţia saltului necondiţionat JUMP µADR se reduce la operaţia:

$$MAR \leftarrow MIR_{11\div0}$$

Deci execuţia acestei microinstrucţiuni constă în încărcarea în MAR a câmpului $MIR_{11\div0}$ din microinstrucţiune (câmpul MICROADRESA DE SALT). Prin urmare, microinstrucţiunea curentă este tocmai o microinstrucţiune de salt simplu (JUMP µADR) iar microadresa de salt (µADR) este codificată în cei mai puţin semnificativi 12 biţi din codul microinstrucţiunii (câmpul $MIR_{11\div0}$).

Pentru ca această microinstrucţiune de salt să fie necondiţionată, trebuie ca funcţia globală de ramificaţie să ia valoarea 1 (g=1), adică condiţia de salt să fie permanent îndeplinită. De pe ultima coloană a tabelului constatăm că $f=\overline{MIR_{12}}$, ceea ce conduce la:

$$g=f \oplus MIR_{12}=\overline{MIR_{12}} \oplus MIR_{12}=1$$

Prin acest artificiu ($f=\overline{MIR_{12}}$) asigurăm g=1, iar g=1 semnifică condiţie de salt îndeplinită (cu alte cuvinte, saltul devine necondiţionat).

Următoarele 12 microinstrucţiuni din tabelul 3.1 sunt salturi simple condiţionate. În cazul acestor microinstrucţiuni apare un succesor dublu, care are forma generică:

SUCCESOR1 else SUCCESOR2

Dacă condiţia testată (pe fals sau pe adevărat) este îndeplinită (dacă g=1) se execută SUCCESOR1, iar dacă condiţia testată este neîndeplinită (dacă g=0) se execută SUCCESOR2. Pentru toate cele 12 microinstrucţiuni în discuţie, SUCCESOR1 este JUMP μADR iar SUCCESOR2 este STEP. După cum s-a arătat mai sus, execuţia succesorului JUMP μADR constă în operaţia:

$$MAR \leftarrow MIR_{11 \div 0}$$

Execuţia succesorului STEP constă în operaţia:

$$MAR \leftarrow MAR+1$$

În concluzie, dacă g=1, se execută saltul la $\mu ADR = MIR_{11 \div 0}$, iar dacă g=0 (dacă condiţia testată este neîndeplinită) se va trece secvenţial la următoarea microinstrucţiune.

B. Successorii JUMPI μADR+INDEX

Successorii JUMPI μADR+INDEX sunt prezentaţi în tabelul 3.2. Execuţia succesorului de salt indexat JUMPI μADR+INDEX constă în:

$$MAR \leftarrow MIR_{11 \div 0} + INDEX \big|_{\text{selectat cu } MIR_{14,13}}$$

Să analizăm acest succesor pe un caz concret. Să presupunem că $MIR_{14,13}$=0 1. Pe baza tabelului 3.3 constatăm că în acest caz INDEX-ul selectat va fi:

$$INDEX_2 = 0\ 0\ 0\ 0\ 0\ IR_{14}\ IR_{13}\ IR_{12}\ IR_{11}\ IR_{10}\ 0\ 0_2$$

Calculul microadresei de salt presupune o operaţie de adunare: se adună indexul $INDEX_2$ la microadresa de bază $MIR_{11 \div 0}$. În binar, această operaţie presupune adunarea a doi operanzi reprezentaţi pe 12 biţi:

MIR_{11}	MIR_{10}	MIR_9	MIR_8	MIR_7	MIR_6	MIR_5	MIR_4	MIR_3	MIR_2	MIR_1	MIR_0	
0	0	0	0	0	IR_{14}	IR_{13}	IR_{12}	IR_{11}	IR_{10}	0	0	+
M	I	C	R	O	A	D	R	E	S	A	D E S A L T	

Pe lângă cei 7 biţi de zero pe care îi conţine, operandul $INDEX_2$ conţine 5 biţi care provin din câmpul OPCODE al instrucţiunii curente (biţii $IR_{14 \div 10}$). Corelând cu modul în care am codificat instrucţiunile din clasa IOP (tabelul 2.2) şi respectiv ID (tabelul 2.3), putem spune că $INDEX_2$ poate lua următoarele valori:

$INDEX_2=000000000000_2=000_H$;dacă în IR se află codul instrucţiunii LD A,*src* (din clasa IOP) sau INC A (din clasa ID)

$INDEX_2=000000000100_2=004_H$;dacă în IR se află codul instrucţiunii LD SP,*src* (din clasa IOP) sau DEC A (din clasa ID)

$INDEX_2=000000001000_2=008_H$;dacă în IR se află codul instrucţiunii LD FLAG,*src* (din clasa IOP) sau NEG A (din clasa ID)

$INDEX_2=000000001100_2=00C_H$;dacă în IR se află codul instrucţiunii ADD A,*src* (din clasa IOP) sau PUSH A (din clasa ID)

.....

$INDEX_2=000001001100_2=04C_H$;dacă în IR se află codul instrucţiunii JNZ *adr* (din clasa IOP) sau IRET (din clasa ID)

$INDEX_2=000001010000_2=050_H$;dacă în IR se află codul instrucţiunii JP *adr* (din clasa IOP)

.....

$INDEX_2=000001100100_2=064_H$;dacă în IR se află codul instrucţiunii CALL *adr* (din clasa IOP)

De aici rezultă că microinstrucţiunea JUMPI $\mu ADR+INDEX_2$, în funcţie de valoarea parametrului $INDEX_2$, va realiza saltul la următoarele microadrese:

$MIR_{11\div0}+000_H$;dacă în IR se află codul instrucţiunii LD A,*src* (din clasa IOP) sau INC A (din clasa ID)

$MIR_{11\div0}+004_H$;dacă în IR se află codul instrucţiunii LD SP,*src* (din clasa IOP) sau DEC A (din clasa ID)

$MIR_{11\div0}+008_H$;dacă în IR se află codul instrucţiunii LD FLAG,*src* (din clasa IOP) sau NEG A (din clasa ID)

$MIR_{11\div0}+00C_H$;dacă în IR se află codul instrucţiunii ADD A,*src* (din clasa IOP) sau PUSH A (din clasa ID)

.....

$MIR_{11\div0}+04C_H$;dacă în IR se află codul instrucţiunii JNZ *adr* (din clasa IOP) sau IRET (din clasa ID)

$MIR_{11\div0}+050_H$;dacă în IR se află codul instrucţiunii JP *adr* (din clasa IOP)

.....

$MIR_{11\div0}+064_H$;dacă în IR se află codul instrucţiunii CALL *adr* (din clasa IOP)

Parametrul $INDEX_2$ conţine 7 biţi de zero şi 5 biţi diferiţi de zero, care provin din câmpul OPCODE al instrucţiunii curente (biţii $IR_{14\div10}$). Teoretic, cei 5 biţi diferiţi de zero pot genera 2^5=32 combinaţii ceea ce înseamnă că parametrul $INDEX_2$ poate lua 32 valori. Asta înseamnă că, teoretic, microinstrucţiunea JUMPI μADR+$INDEX_2$ introduce un punct de ramificaţie în 32 de direcţii. La microprogramul de emulare vom vedea că acest salt indexat va face legătura între microrutina *fetch* operand şi microrutina de execuţie; de pe sfârşitul microrutinei *fetch* operand, prin acest salt indexat, se va sări direct pe începutul microrutinei de execuţie aferentă instrucţiunii curente (aferentă instrucţiunii din IR).

Pe de altă parte observăm că, prin adăugarea celor doi biţi de zero pe poziţiile cele mai puţin semnificative din $INDEX_2$ am obţinut un efect interesant: pentru instrucţiuni succesiv codificate în tabelele 2.2 şi respectiv 2.3, salturile se fac din patru în patru microadrese. Asta înseamnă că am rezervat câte patru locaţii MPM (patru microinstrucţiuni) pentru fiecare microrutină de execuţie. Dacă am fi dorit să rezervăm câte opt locaţii MPM pentru fiecare microrutină de execuţie, ar fi trebuit să concatenăm trei biţi de zero pe poziţiile cele mai puţin semnificative din $INDEX_2$.

Dacă $MIR_{14,13}$=1 0, tot pe baza tabelului 3.3, rezultă că INDEX-ul va avea valoarea:

$$INDEX_3=0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ IR_9\ \ IR_8\ \ 0\ \ 0_2$$

Acest index conţine 10 biţi de zero şi 2 biţi diferiţi de zero. Cei doi biţi diferiţi de zero provin tot din registrul IR (din codul instrucţiunii curente); aceşti doi biţi reprezintă de fapt câmpul MA (mod de adresare) din formatul instrucţiunilor din clasa IOP (vezi figura 2.4 şi tabelul 2.1). Asta înseamnă că microinstrucţiunea JUMPI μADR+$INDEX_3$ introduce un punct de ramificaţie în 4 direcţii (în functie de modul de adresare codificat în instrucţiunea curentă). La microprogramul de emulare vom vedea că această microinstrucţiune va face legătura între microrutina *fetch* instrucţiune şi cele patru microrutine *fetch* operand care vor urma după aceasta; de pe sfârşitul microrutinei *fetch* instrucţiune, cu acest salt indexat, se va sări direct pe începutul microrutinei *fetch* operand aferentă modului de adresare codificat în instrucţiunea curentă.

Din nou am adăugat doi biţi de zero pe poziţiile cele mai puţin semnificative din $INDEX_3$, ceea ce înseamnă că am rezervat câte patru locaţii MPM pentru fiecare microrutină *fetch* operand. Numărul de locaţii MPM (de microinstrucţiuni) rezervate pentru fiecare microrutină se poate aprecia corect doar după proiectarea microrutinelor în discuţie, adică după evaluarea exactă a lungimii acestora.

Utilitatea acestor salturi indexate, cu indecşi proveniţi din *hardware* (biţi din codul instrucţiunii curente) este dificil de intuit de către cititorul nefamiliarizat cu conceptul microprogramării. Ştim deja că un procesor va fi cu atât mai performant (mai rapid) cu cât ciclul instrucţiunii (timpul în care o instrucţiune este procesată) este mai scurt. În cazul microprogramării, procesarea instrucţiunii înseamnă parcurgerea câtorva micrutine: microrutina *fetch* instrucţiune, urmată de microrutina *fetch* operand, care la rândul ei va fi urmată de microrutina de execuţie. Pentru ca ciclul instrucţiunii să fie cât mai scurt, trebuie actionat pe două direcţii:

1. microrutine cât mai scurte

2. timpi de legătură între microrutine cât mai mici

Microinstrucţiunile de salt indexat prezentate mai sus vizează reducerea timpilor de legătură dintre microrutine.

În figura 3.3 sunt prezentate câteva dintre ramificaţiile pe care le vom regăsi în microprogramul de emulare aferent setului de instrucţiuni al procesorului didactic.

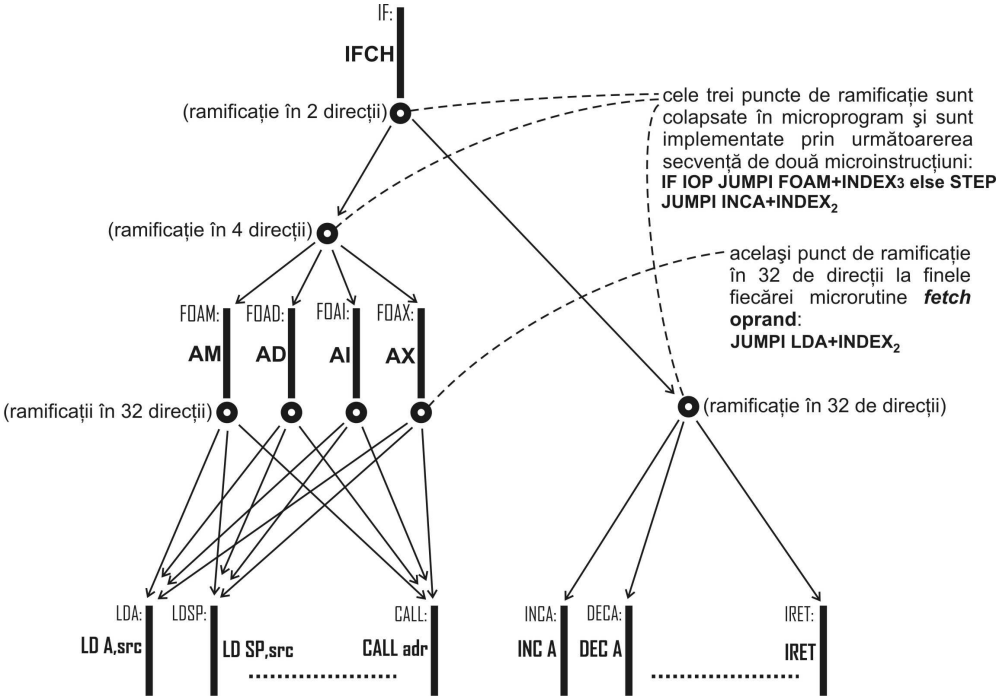


Fig 3.3 Legături între microrutine implementate prin microinstrucţiuni de ramificaţie

Utilitatea salturilor indexate rezultă din această schemă de principiu în care microrutinele sunt reprezentate (simbolic) prin segmente de dreaptă. Microrutinele reprezentate în figura 3.3 sunt:

- **IFCH** -microrutina *fetch* instrucţiune, care începe la eticheta IF:
- **AM** -microrutina *fetch* operand aferentă modului de adresare imediat, (eticheta FOAM:)
- **AD** -microrutina *fetch* operand aferentă modului de adresare direct, (eticheta FOAD:)
- **AI** -microrutina *fetch* operand aferentă modului de adresare indirect, (eticheta FOAI:)
- **AX** -microrutina *fetch* operand aferentă modului de adresare indexat, (eticheta FOAX)

- **LD A,src** -microrutina de execuţie aferentă instrucţiunii LD A,*src*, (eticheta LDA:)
- **LD SP,src** -microrutina de execuţie aferentă instrucţiunii LD SP,*src*, (eticheta LDSP:)
- **CALL adr** -microrutina de execuţie aferentă instrucţiunii CALL *adr*, (eticheta CALL:)
- **INC A** -microrutina de execuţie aferentă instrucţiunii INC A, (eticheta INCA:)
- **DEC A** -microrutina de execuţie aferentă instrucţiunii DEC A, (eticheta DECA:)
- **IRET** -microrutina de execuţie aferentă instrucţiunii IRET, (eticheta IRET:)

Microrutina *fetch* instrucţiune (**IFCH**) are sarcina de a citi codul instrucţiunii din memorie şi de a încărca acest cod în registrul IR; instrucţiunea încărcată în IR va deveni astfel instrucţiune curentă. La ieşirea din microrutina **IFCH** apare un punct de ramificaţie în 2 direcţii: IOP şi respectiv ID. Se va merge pe direcţia IOP dacă instrucţiunea curentă este din clasa IOP; dacă instrucţiunea curentă este din clasa ID se va merge în continuare pe direcţia ID. Pe direcţia IOP urmează un nou punct de ramificaţie în 4 direcţii, prin care se stabilesc legăturile la cele 4 microrutine *fetch* operand aferente celor 4 moduri de adresare (**AM**, **AD**, **AI**, şi **AX**). Se va face evident saltul (legătura) la microrutina *fetch* operand aferentă modului de adresare codificat în instrucţiunea curentă (în funcţie de valorile biţilor IR_9 şi IR_8 din codul instrucţiunii curente, vezi figura 2.4 şi tabelul 2.1) . Pe direcţia ID urmează un nou punct de ramificatţie în 32 direcţii (practic doar 20 utilizate). Acest punct de ramificaţie implementează legăturile la microrutinele de execuţie aferente celor 20 instrucţiuni din clasa ID (tabelul 2.3). Se va face evident saltul la microrutina de execuţie aferentă instrucţiunii curente (ramificaţie în funcţie de valoarea câmpului OPCODE din instrucţiunea curentă, vezi figura 2.6 şi tabelul 2.3). Cele trei puncte de ramificaţie (în 2, în 4 şi respectiv în 32 direcţii) vor fi implementate în microprogramul de emulare prin următoarea secventă de 2 microinstrucţiuni (figura 3.3):

IF IOP JUMPI FOAM+(IR_9,IR_8,0,0) else STEP
JUMPI INCA+($IR_{14\div10}$,0,0)

În cele două microinstrucţiuni am folosit următoarele notaţii (vezi tabelul 3.3):

$INDEX_2$=($IR_{14\div10}$,0,0) ;index care include câmpul OPCODE din instrucţiunea ;curentă

$INDEX_3$=(IR_9,IR_8,0,0) ;index care include câmpul MA din instrucţiunea ;curentă

La sfârşitul celor 4 microrutine *fetch* operand (**AM**, **AD**, **AI**, **AX**) sunt implementate 4 puncte de ramificaţie identice. Sunt puncte de ramificaţie în 32 de direcţii (practic doar 26 utilizate), prin care se stabilesc legăturile la microrutinele de execuţie aferente celor 26 instrucţiuni din clasa IOP (tabelul 2.2). Aceste patru puncte de ramificaţie (identice) vor fi implementate în microprogramul de emulare prin microinstrucţiunea de salt indexat:

JUMPI LDA+($IR_{14\div10}$,0,0)

Se va face evident saltul la microrutina de execuţie aferentă instrucţiunii curente care este desigur o instrucţiune din clasa IOP; aceasta este motivaţia saltului indexat cu un index ($INDEX_2$) care include câmpul OPCODE din codul instrucţiunii curente (vezi fig. 2.4 şi tabelul 2.2).

Este evident faptul că utilizarea salturilor indexate conduce la optimizarea microcodului (legături rapide între microrutine). Ramificaţiile indexate prezentate schematic în figura 3.3 le vom regăsi în microprogramul de emulare a setului de instrucţiuni aferent procesorului didactic (paragraful 3.8.2).

Schema bloc din figura 3.1 ilustreză structura unităţii de control microprogramate; utilizând această schemă se poate explica principial funcţionarea unităţii microprogramate. Dacă dorim să fim riguroşi, introducerea salturilor indexate reclamă o ușoară ajustare a schemei bloc din figura 3.1; varianta ajustată este prezentată în figura 3.4.

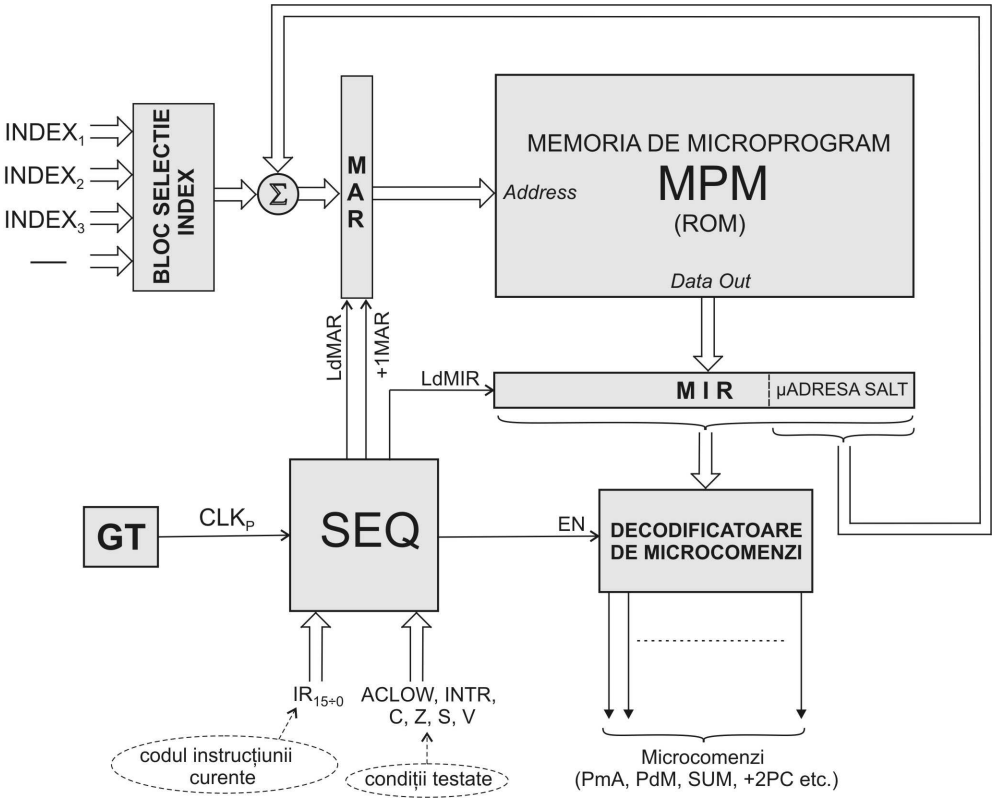


Fig 3.4 Strctura unităţii de control microprogramate cu evidenţierea mecanismului de construcţie a microadreselor de salt

În raport cu schema din figura 3.1, în figura 3.4 apare suplimentar un sumator şi un bloc de selecţie index care generează indecşii definiţi în microinstrucţiunea de ramificaţie. În tabelul 3.3 am definit trei indecşi ($INDEX_1$, $INDEX_2$ şi $INDEX_3$). Aceşti 3 indecşi sunt aplicaţi pe intrările blocului de selecţie index din figura 3.4. La salturile indexate (succesorii JUMPI din tabelul 3.2), INDEX-ul selectat de către blocul de selecţie index, se adună la microadresa de salt preluată din registrul MIR (la câmpul $MIR_{11\div0}$) şi rezultatul se încarcă în registrul MAR (devenind astfel **microadresa reală de salt**). Selecţia INDEX-ului se realizează cu ajutorul câmpului $MIR_{14,13}$ (câmpul SELECŢIE INDEX din microinstrucţiunea curentă). Blocul de selecţie index poate fi implementat cu ajutorul unui set de multiplexoare 4:1; proiectarea acestui bloc va fi realizată în cadrul paragrafului 3.7.2. Dacă blocul de selecţie index selectează un index de valoare zero, atunci în MAR se va încărca chiar câmpul $MIR_{11\div0}$ (câmpul MICROADRESA DE SALT) din microinstrucţiunea curentă. În această manieră, salturile indexate (JUMPI) se transformă în salturi simple, neindexate (JUMP).

Implementarea în *hardware* a funcţiei globale de ramificaţie (g) se va realiza pe baza condiţiilor sintetizate în tabelul 3.2. Aceste condiţii sunt identice cu cele sintetizate în tabelul 3.1 deoarece tabelul 3.1 reprezintă un caz particular al tabelului 3.2 (dacă indexul selectat are valoarea zero atunci succesorii JUMPI din tabelul 3.2 se transformă în succesorii echivalenţi JUMP din tabelul 3.1). Implementarea funcţiei globale de ramificaţie va fi prezentată la 3.7.3.

3.4.2 Formatul microinstrucţiunii de transfer

A. Formatul microinstrucţunii de transfer sincron

Microinstrucţiunea de transfer sincron va genera comenzi elementare (C_L) prin care va controla transferurile de informaţii între registrele procesorului. După cum am arătat în capitolul 2, aceste transferuri de informaţii între registre sunt sincronizate cu semnalul de tact al procesorului şi se execută pe durata unei singure perioade de tact; de aici şi denumirea de **„microinstrucţiune de transfer sincron”**. Formatul microinstrucţiunii de transfer sincron este prezentat în figura 3.5.a).

După cum am văzut la 3.4.1, bitul MIR_{19} specifică tipul microinstrucţiunii:

MIR_{19}=0 -defineşte microinstrucţiunea de ramificaţie

MIR_{19}=1 -defineşte microinstrucţiunea de transfer (de transfer sincron sau asincron)

În cazul microinstrucţiunilor de transfer, următorul bit (MIR_{18}) defineşte tipul microinstrucţiunii de transfer:

MIR_{18}=0 -defineşte microinstrucţiunea de transfer sincron

MIR_{18}=1 -defineşte microinstrucţiunea de transfer asincron

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	**0**	**SURSA SBUS**			**SURSA DBUS**			**OPERAŢIE ALU**			**SURSA RBUS**			**DESTINAŢIE RBUS**			**„OTHER OPERATIONS"**		
		NONE			NONE			SBUS			NONE			NONE			NOP		
		PdIR[OP]			PdM			$\overline{SBUS}$			PdALU			PmIR			(Cin,PdCOND)		
		PdIR[IND]			Pd $\overline{M}$			DBUS			PdIVR			PmA			(INTA,-2SP)		
		PdA			PdRG			SUM			PdFLAG			PmSP			PdCOND		
		PdSP			Pd $\overline{RG}$			AND			Pd ADR			PmM			Cin		
		PdOs			PdPC			OR			–			PmRG			PmFLAG		
		Pd-1s			PdOd			XOR			–			PmPC			+2SP		
		–			Pd-1d			INV			–			PmADR			–2SP		

A. Formatul microinstrucţiunii de transfer sincron

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	**1**	**SURSA SBUS**			**SURSA DBUS**			**OPERAŢIE ALU**			**SURSA RBUS**			**MEMORY OPERATION**		**SHIFT & OTHER OPERATIONS**			
		NONE			NONE			SBUS			NONE			NOP		NOP			
		PdIR[OP]			PdM			$\overline{SBUS}$			PdALU			IFCH		DSA			
		PdIR[IND]			Pd $\overline{M}$			DBUS			PdIVR			READ		DDA			
		PdA			PdRG			SUM			PdFLAG			WRITE		RS			
		PdSP			Pd $\overline{RG}$			AND			Pd ADR					RD			
		PdOs			PdPC			OR			–					RSC			
		Pd-1s			PdOd			XOR			–					RDC			
		–			Pd-1d			–			–					DDL			
																+2PC			
																A(I)BVI			
																A(O)BVI			
																A(I)BEO			
																A(I)BEI			
																(A(O)BI, A(O)BE)			
																A(O)BPO			
																PdCOND			

B. Formatul microinstrucţiunii de transfer asincron

Fig 3.5 Formatul microinstrucţiunii de transfer

În următoarele câmpuri sunt codificate comenzi logice elementare (C_L) care, în contextul microprogramării, se numesc microcomenzi. În cele ce urmează vom utiliza frecvent termenul microcomandă. Trebuie să înţelegem că acest termen este uzual atunci când se proiectează o unitate de control microprogramată. Pe de altă parte rebuie să realizăm că, microcomanda de la unitatea de control microprogramată este identică cu comanda logică elementară de la unitatea de control cablată.

Vom parcurge de la stânga la dreapta câmpurile din formatul microinstrucţiunii de transfer sincron (figura 3.5) şi vom explica comenzile codificate în aceste câmpuri. Câmpul SURSA SBUS (biţii $MIR_{17\div15}$) selectează sursa ce va fi emisă pe SBUS (registrul al cărui conţinut va fi emis pe SBUS). Operaţiile realizate de microcomenzile codificate în acest câmp sunt sintetizate în tabelul 3.4.

MIR_{17}	MIR_{16}	MIR_{15}	Comanda	Sursa pe SBUS	
0	0	0	NONE	(nici-o sursă emisă pe SBUS)	
0	0	1	PdIR[OP]	SBUS←$IR_{7\div0}$	(operandul imediat emis pe SBUS)
0	1	0	PdIR[IND]	SBUS←$IR_{3\div0}$	(indexul emis pe SBUS)
0	1	1	PdA	SBUS←A	(Acumulatorul emis pe SBUS)
1	0	0	PdSP	SBUS←SP	(*Pointer* –ul de stivă emis pe SBUS)
1	0	1	Pd0s	SBUS←0	(constanta zero emisă pe SBUS)
1	1	0	Pd-1s	SBUS←–1	(constanta minus unu emisă pe SBUS)
1	1	1	–	–	

Tabelul 3.4 Selecţia sursei pe SBUS

Din tabelul 3.4 rezultă că, pe lângă registrele conectate la SBUS (vezi schema bloc a procesorului didactic din figura 2.2), pe SBUS mai pot fi emise şi două constante: zero şi respectiv minus unu.

Într-o manieră similară, câmpul SURSA DBUS (biţii $MIR_{14\div12}$) selectează sursa pe DBUS (registrul al cărui conţinut va fi emis pe DBUS). Operaţiile realizate de microcomenzile codificate în acest câmp sunt sintetizate în tabelul 3.5.

Să observăm că pe DBUS pot fi emise registrele care sunt conectate la acest bus în schema bloc din figura 2.2. Să mai observăm că registrul general selectat cu câmpul ADR.REG. din codul instrucţiunii curente (vezi fig. 2.5) poate fi emis şi inversat (bit cu bit) pe DBUS (complementul faţă de unu al operandului memorat în registru). De asemenea şi registrul M poate fi emis inversat pe DBUS (complementul faţă de unu al operandului din M). Pe de altă parte, pe DBUS pot fi emise constantele zero şi respectiv minus unu (ca şi pe SBUS).

MIR_{14}	MIR_{13}	MIR_{12}	Comanda	Sursa pe DBUS	
0	0	0	NONE	(nici-o sursă emisă pe DBUS)	
0	0	1	PdM	DBUS←M	(registrul M emis pe DBUS)
0	1	0	Pd $\overline{M}$	DBUS← $\overline{M}$	(M, inversat bit cu bit, emis pe DBUS)
0	1	1	PdRG	DBUS←RG	(RG selectat de câmpul ADR.REG. din IR)
1	0	0	Pd $\overline{RG}$	DBUS← $\overline{RG}$	(registrul general, inversat bit cu bit)
1	0	1	PdPC	DBUS←PC	(registrul *Program Counter* emis pe DBUS)
1	1	0	Pd0d	DBUS←0	(constanta zero emisă pe DBUS)
1	1	1	Pd-1d	DBUS←–1	(constanta minus unu emisă pe DBUS)

Tabelul 3.5 Selecţia sursei pe DBUS

Câmpul OPERAŢIE ALU (biţii $MIR_{11\div9}$) selectează operaţia ALU aplicată celor 2 operanzi (preluaţi de pe SBUS şi respectiv DBUS). Prin decodificarea acestui câmp se vor genera microcomenzile pentru unitatea ALU. Operaţiile realizate de microcomenzile codificate în acest câmp sunt sintetizate în tabelul 3.6.

MIR_{11}	MIR_{10}	MIR_9	Comanda	Operaţia ALU
0	0	0	SBUS	(ALU trnsmite operandul preluat de pe SBUS, nemodificat)
0	0	1	$\overline{SBUS}$	(ALU trnsmite operandul preluat de pe SBUS, inversat bit cu bit)
0	1	0	DBUS	(ALU trnsmite operandul preluat de pe DBUS, nemodificat)
0	1	1	SUM	(ALU efectuează operaţia de adunare)
1	0	0	AND	(ALU efectuează operaţia SI Logic)
1	0	1	OR	(ALU efectuează operaţia SAU Logic)
1	1	0	XOR	(ALU efectuează operaţia SAU Exclusiv)
1	1	1	–	–

Tabelul 3.6 Codificarea comenzilor pentru ALU (operaţiile ALU)

Câmpul SURSA RBUS (biţii $MIR_{8\div6}$) selectează sursa ce va fi emisă pe RBUS. Operaţiile realizate de microcomenzile codificate în acest câmp sunt sintetizate în tabelul 3.7.

MIR_8	MIR_7	MIR_6	Comanda	Sursa pe RBUS	
0	0	0	NONE	(nici-o sursă)	
0	0	1	PdALU	RBUS←ALU	(rezultatul ALU emis pe RBUS)
0	1	0	PdIVR	RBUS←IVR	(registrul IVR emis pe RBUS)
0	1	1	PdFLAG	RBUS←FLAG	(registrul FLAG emis pe RBUS)
1	0	0	Pd ADR	RBUS←ADR	(registrul ADR emis pe RBUS)
1	0	1	–	–	
1	1	0	–	–	
1	1	1	–	–	

Tabelul 3.7 Selecţia surselor pe RBUS

După cum se observă din tabelul 3.7, pe RBUS pot emite informaţii 4 surse: registrele IVR, FLAG şi ADR precum şi unitatea ALU (rezultatul obţinut pe ieşirile ALU). Aceste surse sunt evidenţiate şi în schema bloc a procesorului didactic (figura 2.2). Microomanda NONE înseamnă „nici-o sursă” selectată pentru emisie pe RBUS.

Câmpul DESTINAŢIE RBUS (biţii $MIR_{5\div3}$) selectează registrul destinaţie în care urmează a fi încărcate datele de pe RBUS. În acest câmp sunt codificate microcomenzi de tipul „primeşte registru”; vor fi desigur codificate comenzi „primeşte” pentru toate registrele conectate ca destinaţii pe RBUS. Comenzile codificate în câmpul DESTINAŢIE RBUS şi respectiv operaţiile realizate de aceste comenzi sunt prezentate în tabelul 3.8

MIR_5	MIR_4	MIR_3	Comanda	Operaţia realizată	
0	0	0	NONE	(nici-o destinaţie)	
0	0	1	PmIR	IR ←RBUS	(încărcare date de pe RBUS în IR)
0	1	0	PmA	A ←RBUS	(încărcare date de pe RBUS în A)
0	1	1	PmSP	SP ←RBUS	(încărcare date de pe RBUS în SP)
1	0	0	PmM	M ←RBUS	(încărcare date de pe RBUS în M)
1	0	1	PmRG	RG ←RBUS	(încărcare date de pe RBUS în RG)
1	1	0	PmPC	PC ←RBUS	(încărcare date de pe RBUS în PC)
1	1	1	PmADR	ADR←RBUS	(încărcare date de pe RBUS în ADR)

Tabelul 3.8 Selecţia registrului ce va prelua detele de pe RBUS

În câmpul „OTHER OPERATIONS” (biţii $MIR_{2\div0}$) am codificat comenzi care declanşează diverse operaţii. Fiind operaţii foarte diverse ca tip şi acţiune, le-am etichetat

drept „alte operaţii”. De aici rezidă şi ideia de a eticheta acest ultim câmp din componenţa microinstrucţiunii de transfer sincron cu denumirea „OTHER OPERATIONS”. Operaţiile codificate în acest câmp sunt prezentate în tabelul 3.9.

MIR_2	MIR_1	MIR_0	Comanda	Operaţia realizată
0	0	0	NOP	NOP (nici-o operaţie)
0	0	1	(C_{in},PdCOND)	C_{in}←1; C/Z/S/V←C_{out}/Z_R/S_R/DCR
0	1	0	(INTA,-2SP)	INTA=1; SP←SP-2 (achitare INTR şi decrementare SP)
0	1	1	PdCOND	C/Z/S/V←Cout/Z_R/S_R/DCR (poziţionare *flag*-uri)
1	0	0	C_{in}	C_{in} ←1 (1 aplicat pe intrarea C_{in} la ALU)
1	0	1	PmFLAG	FLAG←$RBUS_{3\div0}$ (încărcare date de pe RBUS în FLAG)
1	1	0	+2SP	SP←SP+2 (incrementare *pointer* de stivă)
1	1	1	–2SP	SP←SP–2 (decrementare *pointer* de stivă)

Tabelul 3.9 Operaţiile codificate în câmpul „OTHER OPERATIONS”

Cu ajutorul microinstrucţiunii de transfer sincron se pot controla următoarele tipuri de operaţii:

1. transferul datelor dintr-un registru sursă într-un registru destinaţie.

   Exemple:

   a) microinstrucţiunea A $\overline{SBUS}$ NONE, PdALU, PmM realizează operaţia M← $\overline{A}$.

   b) microinstrucţiunea NONE DBUS PC, PdALU, PmADR realizează operaţia ADR←PC.

   Observaţie:
   După cum se observă din cele două exemple de mai sus, pentru specificarea microinstrucţiunilor, am adoptat un limbaj bazat pe mnemonice. Mnemonica unei microinstrucţiuni de transfer sincron (care controlează efectiv şi o operaţie de transfer şi nu doar una de tip „Other Operations”), începe cu o expresie de tipul «SursaSBUS OperaţieALU SursaDBUS». Această expresie cu trei termeni va clarifica comenzile codificate în primele trei câmpuri ale microinstrucţiunii. De exemplu, expresia NONE DBUS PC din microinstrucţiunea exemplificată la punctul b) de mai sus, specifică următoarele comenzi:

   - *comanda NONE codificată în câmpul SURSA SBUS ($MIR_{17\div15}$=000)*
   - *comanda PdPC codificată în câmpul SURSA DBUS ($MIR_{14\div12}$=101)*

- *comanda DBUS codificată în câmpul OPERAŢIE ALU* ($MIR_{11\div9}=010$)

2. transferuri de date între registre cu procesarea în ALU a informaţiilor transferate

   Exemple:

   a) microinstrucţiunea A AND M, PdALU, PmA realizează operaţia A←A•M.

   b) microinstrucţiunea A SUM M, PdALU, PmA, PdCOND realizează adunarea A←A+M şi poziţionarea *flag*-urilor de condiţii din registrul FLAG în acord cu rezultatul adunării.

   c) microinstrucţiunea A SUM $\overline{M}$, PdALU, PmA, (Cin,PdCOND) realizează scăderea A←A–M şi poziţionarea *flag*-urilor de condiţii din registrul FLAG în acord cu rezultatul scăderii (scădere se realizează prin adunarea la descăzut a complementului faţă de 2 al scăzătorului).

   d) microinstrucţiunea A SUM NONE, PdALU, PmA, (Cin,PdCOND) realizează adunarea A←A+1 şi poziţionarea *flag*-urilor de condiţii din registrul FLAG în acord cu rezultatul adunării.

3. alte operaţii

   Exemple:

   a) microinstrucţiunea +2SP realizează operaţia SP←SP+2 (incrementarea cu 2 a pointerului de stivă).

   b) microinstrucţiunea (ITRACK,–2SP) realizează achitarea întreruperii şi operaţia SP←SP–2 (decrementarea cu 2 a pointerului de stivă).

   Observaţie:
   Dacă nu produc conflicte pe busuri sau la nivelul registrelor, comenzile de tipul „Other Operations" pot acţiona şi concurent cu o operaţie de transfer sincron. De exemplu, microinstrucţiunea: A XOR M, PdALU, PmA, +2SP realizează două operaţii:

   - *A←A⊕M (transfer sincron cu procesare în ALU)*
   - *SP←SP+2 (incrementare cu 2 a pointerului de stivă; operaţie „locală" care afectează doar registrul SP, implementat în variantă „numărător reversibil")*

   Cele două operaţii pot fi codificate în aceeaşi microinstrucţiune (pot fi concurente) deoarece nu generează conflicte în structura procesorului didactic (vezi figura 2.2).

Să remarcăm apariţia comenzii PdCOND care nu exista la unitatea de control cablată. Această comandă realizează setarea *flag*-urilor de condiţii din registrul FLAG în acord cu

rezultatul obţinut pe ieşirea ALU (realizează operaţia C/Z/S/V←Cout/Z_R/S_R/DCR). Prin urmare, microcomanda PdCOND realizează poziţionarea eficientă a *flag*-urilor de condiţii, este necesară în microrutinele de execuţie aferente instrucţiunilor aritmetice şi revendică o implementare adecvată în *hardware*.

În final să consemnăm că **succesorul implicit al microinstrucţiunii de transfer sincron este STEP**. Fiind un succesor implicit, el nu a mai fost codificat în microinstrucţiune.

B. Formatul microinstrucţunii de transfer asincron

Microinstrucţiunea de transfer asincron va genera comenzi elementare (microcomenzi) prin care va controla transferurile de informaţii procesor-memorie (va controla operaţiile de *fetch* instrucţiune, *fetch* operand şi de scriere în memorie). În aceste operaţii de transfer sunt implicate două blocuri conectate la busul sistemului de calcul (figura 2.1), procesorul şi respectiv memoria principală. După cum am arătat în capitolul 2, memoria este în general mai lentă decât procesorul (o citire sau o scriere în memorie se „întinde" pe mai multe perioade de tact procesor). Ştim deja că, în astfel de situaţii, dispozitivul care va impune ritmul este întotdeauna dispozitivul mai lent (în cazul nostru memoria). Mai ştim că aceste transferuri sunt de regulă controlate într-o manieră asincronă, utilizând semnale de *hand-shaking* (figura 2.14); de aici şi denumirea de **„microinstrucţiune de transfer asincron"**. Formatul microinstrucţiunii de transfer asincron este prezentat în figura 3.5.B.

Biţii $MIR_{19,18}$=11 definesc microinstrucţiunea de transfer asincron. Următoarele patru câmpuri (SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS) sunt preluate fără nici o morificare din microinstrucţiunea de transfer sincron (figura 3.5.A). În microinstrucţiunea de transfer asincron cele patru câmpuri menţionate au semnificaţie numai când în câmpul MEMORY OPERATION este codificată microcomanda WRITE; aceste câmpuri au rolul de a emite pe RBUS datele de scris în memorie. Codificarea identică a celor patru câmpuri (în cele două tipuri de microinstrucţiuni de transfer) va conduce la simplificări în *hardware* (se simplifică decodificatoarele de microcomenzi aferente acestor câmpuri).

În câmpul MEMORY OPERATION (biţii $MIR_{5,4}$) sunt codificate microcomenzile aferente operaţiilor de transfer cu memoria. Aceste comenzi şi operaţiile aferente sunt prezentate în tabelul 3.10

MIR_5	MIR_4	Comanda	Operaţia realizată	
0	0	NOP	NOP	(nici-o operaţie cu memoria)
0	1	IFCH	IR←$MEM\|_{ADR}$	(*fetch* instrucţiune)
1	0	READ	M←$MEM\|_{ADR}$	(*fetch* date din memorie)
1	1	WRITE	$MEM\|_{ADR}$←RBUS	(scriere date în memorie)

Tabelul 3.10 Comenzile pentru memorie şi operaţiile aferente

NOP însemnă nici o operaţie cu memoria. Se impune o astfel de posibilitate deoarece, în microprogramul de emulare, ar putea apărea necesitatea unei microinstrucţiuni de transfer asincron care activează doar o microcomandă (doar o operaţie) de tip SHIFT & OTHER OPERATIONS şi care nu lansează nici-o operaţie în memorie.

IFCH reprezintă o abreviere a expresiei ***I**nstruction **F**et**ch***. Prin urmare, această microcomandă va realiza *fetch*-ul instrucţiunii din memorie (conţinutul locaţiei de memorie selectată cu adresa existentă în registrul ADR este citit şi încărcat în registrul IR).

READ reprezintă microcomanda destinată citirii unui operand din memorie. Prin urmare, această microcomandă va declanşa operaţia de citire a datelor memorate în locaţia selectată de adresa existentă în registrul ADR; datele citite din memorie vor fi încărcate în registrul M.

Să observăm că microcomenzile IFCH şi READ fac aceeaşi operaţie în memorie (citirea informaţiei stocate în locaţia selectată de adresa existentă în registrul ADR). Cele două microcomenzi diferă numai prin faptul că specifică destinaţii diferite pentru informaţia citită din memorie. IFCH încarcă informaţia citită din memorie în registrul IR, iar READ în registrul M. În mod cert vom avea o microinstrucţiune de transfer asincron cu microcomanda IFCH codificată în cadrul ei, în microrutina *fetch* instrucţiune şi vom avea microinstrucţiuni cu READ în microrutinele *fetch* operand aferente modurilor de adresare care specifică operand în memorie (modurile AI şi respectiv AX). Introducerea a două microcomenzi de citire din memorie (una pentru *fetch*-ul instrucţiunii şi cealaltă tentru *fetch*-ul operandului) s-a făcut din considerente de performanţă. Să mai menţionăm că operaţiile de citire din memorie declanşate de microcomenzile IFCH şi READ vor fi controlate efectiv de automatul SEQ (figura 3.1). Detalierile necesare vor fi făcute ulterior (în paragraful referitor la proiectarea SEQ).

WRITE reprezintă microcomanda de scriere în memorie; datele prezente pe RBUS vor fi scrise în locaţia de memorie selectată de adresa existentă în ADR. Prin urmare, pentru ca operaţia de scriere să se deruleze corect, pe toată durata acestei scrieri, trebuie asigurate (emise) pe RBUS datele de scris. Microcomenzile codificate în cele patru câmpuri (SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS), care au semnificaţie doar dacă în microinstrucţiune (în câmpul MEMORY OPERATION) este codificată o microcomandă WRITE, vor realiza acest lucru (vor emite datele de scris pe RBUS).

În câmpul SHIFT & OTHER OPERATIONS (biţii $MIR_{3\div0}$) am codificat o microcomandă NOP, 7 microcomenzi de deplasare/rotire şi 8 microcomenzi de tip *„other”*. Aceste microcomenzi şi operaţiile aferente sunt prezentate în tabelul 3.11.

Micromenzile de deplasare şi rotire vor avea rol în execuţia instrucţiunilor de deplasare şi rotire. Să consemnăm ca am utilizat mnemonice diferite pentru microcomenzile de deplasare şi rotire în raport cu mnemonicile utilizate pentru instrucţiunile de deplasare şi rotire. De exemplu, microcomanda DDA (deplasare dreapta aritmetică) va fi activată în microrutina de execuţie aferentă instrucţiunii ASR A (*Arithmetic Shift Right*). Utilizarea de mnemonici diferite pentru microcomandă şi respectiv pentru instrucţiunea vizată de această microcomandă are drept scop evitarea confuziilor.

Microcomenzile de deplasare şi rotire (DDA, DSA, RD, RS, RDC, RSC, DDL) acţionează asupra registrului acumulator şi ca mod de operare se conformează schemelor de principiu din figura 2.60. În principiu, conţinutul acumulatorului este deplasat serial cu o poziţie în sensul specificat de microcomandă (la stânga sau la dreapta).

MIR_3	MIR_2	MIR_1	MIR_0	Comanda	Operaţia realizată
0	0	0	0	NOP	NOP (nici-o operaţie de tip *shift & other)*
0	0	0	1	DDA	DrA, $IS_{15}=A_{15}$, $C \leftarrow A_0$ (deplasare aritmetică la dreapta)
0	0	1	0	DSA	StA, $IS_0=0$, $C \leftarrow A_{15}$ (deplasare aritmetică la stânga)
0	0	1	1	RD	DrA, $IS_{15}=A_0$, $C \leftarrow A_0$ (rotire la dreapta)
0	1	0	0	RS	StA, $IS_0=A_{15}$, $C \leftarrow A_{15}$ (rotire la stânga)
0	1	0	1	RDC	DrA, $IS_{15}=C$, $C \leftarrow A_0$ (rotire la dreapta cu *Carry*)
0	1	1	0	RSC	StA, $IS_0=C$, $C \leftarrow A_{15}$ (rotire la stânga cu *Carry*)
0	1	1	1	DDL	DrA, $IS_{15}=0$, $C \leftarrow A_0$ (deplasare logică la dreapta)
1	0	0	0	+2PC	PC←PC+2 (incrementare PC)
1	0	0	1	A(1)BVI	BVI←1 (setare bistabil de validare întreruperi)
1	0	1	0	A(0)BVI	BVI←0 (resetare bistabil de validare întreruperi)
1	0	1	1	A(1)BE0	BE0←1 (setare bistabil de excepţie 0)
1	1	0	0	A(1)BE1	BE1←1 (setare bistabil de excepţie1)
1	1	0	1	(A(0)BI,A(0)BE)	BI/BE←0 (resetare bistabili de întreruperi şi excepţii)
1	1	1	0	A(0)BPO	BPO←0 (resetare bistabil de pornire-oprire)
1	1	1	1	PdCOND	$C/Z/S/V \leftarrow Cout/Z_R/S_R/DCR$ (poziţionare *flag*-uri)

IS_{15} –intrarea serială în rangul A15 al registrului A (fig 2.59)
IS_0 –intrarea serială în rangul A0 al registrului A (fig 2.59)

Tabelul 3.11 Comenzile codificate în câmpul SHIFT & OTHER OPERATIONS

Pe de altă parte, aceste microcomenzi vor avea o acţiune specială asupra intrărilor seriale ale registrului A (IS_{15} şi respectiv IS_0) şi asupra *flag*–ului C din registrul FLAG (vezi figura 2.60). Prin intermediul intrărilor seriale IS_{15} şi IS_0 se acţionează asupra biţilor extremi din A (A_{15} şi respectiv A_0). Acţiunile acestor microcomenzi asupra intrărilor seriale IS_{15} şi respectiv IS_0, precum şi asupra *flag*-ului C din registrul FLAG, sunt sintetizate în figura 3.6. Schema din figura 3.6 s-a elaborat pe baza operaţiilor sintetizate în tabelul 3.11 iar operaţiile din acest tabel au fost sintetizate pe baza schemelor de principiu din figura 2.60. De exemplu, pe baza tabelului 3.11 se pot scrie următoarele ecuaţii:

$$\begin{aligned} IS_{15} &= A_{15} \bullet DDA + A_0 \bullet RD + C \bullet RDC \\ IS_0 &= A_{15} \bullet RS + C \bullet RSC \end{aligned} \qquad (3.2)$$

Ecuaţiile (3.2) au fost implementate în schema din figura 3.6.

Pe baza schemelor de principiu din figura 2.60 şi a operaţiilor/comenzilor sintetizate în tabelul 3.11, se pot sintetiza şi ecuaţiile comenzilor de deplasare stânga-dreapta dedicate registrului acumulator (DrA şi respectiv StA):

$$\begin{aligned} DrA &= DDA+RD+RDC+DDL \\ StA &= DSA+RS+RSC \end{aligned} \qquad (3.3)$$

Implementând în *hardware* ecuaţiile (3.3) se obţin deci comenzile de deplasare stânga-dreapta dedicate registrului acumulator (fig 3.6).

Dacă am decis implementarea *flag*-ului C printr-un bistabil J-K *Master-Slave* atunci, tot pe baza operaţiilor sintetizate în tabelul 3.11, se pot elabora ecuaţiile întrărilor J_C şi respectiv K_C:

$$J_C=A_0 \bullet DrA+A_{15} \bullet StA \qquad (3.4)$$

Flag-ul C este afectat însă şi de instrucţiunile LD FLAG,*src* şi respectiv IRET care, în faza (microrutina) de execuţie, activează microcomanda PmFLAG; comanda PmFLAG trebuie să încarce în *flag*-ul C starea liniei $RBUS_3$.

De asemenea *flag*-ul C este afectat şi de instrucţiunile aritmetice care, în microrutina de execuţie, activează microcomanda PdCOND; microcomanda PdCOND trebuie să încarce în *flag*-ul C condiţia C_{out} (de la ALU).

Dacă, în ecuaţia (3.4), adăugăm şi aceste ultime două acţiuni asupra *flag*-ului C, obţinem:

$$J_C=A_0 \bullet DrA+A_{15} \bullet StA+RBUS_3 \bullet PmFLAG+C_{out} \bullet PdCOND \qquad (3.5)$$

Ecuaţia (3.5) a fost implementată de asemenea în figura 3.6.

Se ştie că, la bistabilul J-K, intrarea K_C îndeplineşte condiţia: $K_C=\overline{J_C}$. Prin urmare, inversorul din figura 3.6 generează semnalul pentru intrarea K_C, pornind de la semnalul aplicat pe intrarea J_C.

În sfârşit, instrucţiunile care afectează (setează/resetează) *flag*-ul *Carry* sunt:

- instrucţiunile de deplasare şi respectiv rotire la dreapta (ASR A, LSR A, ROR A şi RRC A) care în microrutina de execuţie activează microcomenzile de deplasare/rotire la dreapta: DDA, DDL, RD şi RDC. În *hardware*, din aceste 4 microcomenzi se obţine comanda DrA.

- instrucţiunile de deplasare şi respectiv rotire la stânga (ASL A, ROL A şi RLC A) care în microrutina de execuţie activează microcomenzile de deplasare/rotire la stânga: DSA, RS şi RSC. În *hardware*, din aceste 4 microcomenzi se obţine comanda StA.

- instrucţiunile LD FLAG,*src* şi respectiv IRET (care în microrutina de execuţie, activează comanda PmFLAG

- instrucţiunile aritmetice care, în microrutina de execuţie, activează comanda PdCOND

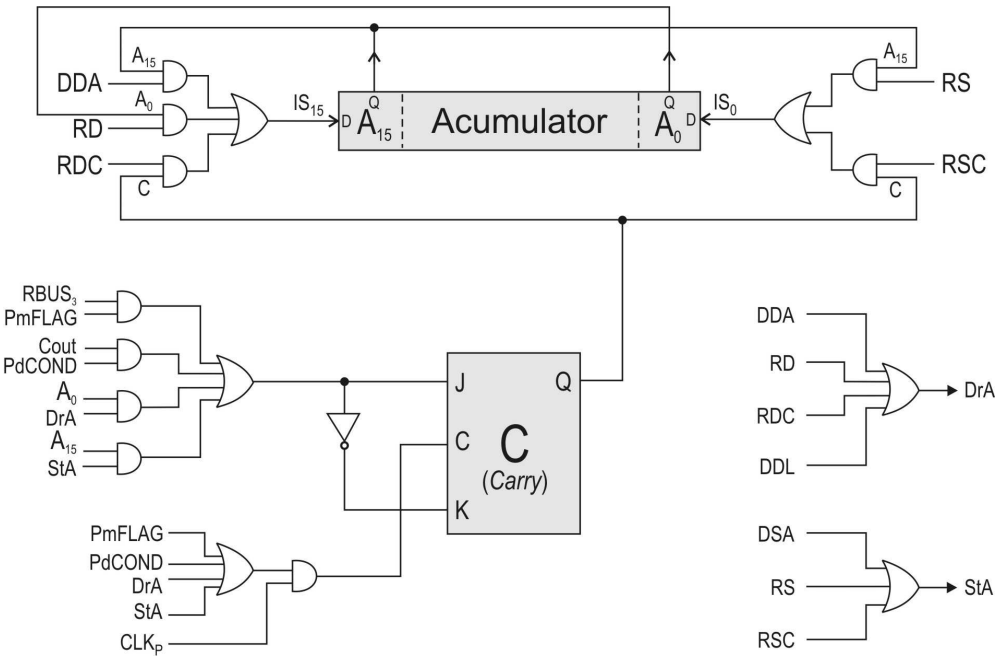


Fig 3.6 Acţiunile microcomenzilor de deplasare şi rotire asupra biţilor extremi din A şi asupra *flag*-ului *Carry*

Poziţionarea *flag*-ului C înseamnă de fapt setarea sau resetarea sa printr-o operaţie de încărcare paralelă, sincronizată cu semnalul de tact al procesorului (CLK_P). În baza acestui raţionament, rezultă ecuaţia semnalului de tact pentru *flag*-ul (bistabilul) C:

$$C_C=(PmFLAG+PdCOND+DrA+StA)\bullet CLK_P \qquad (3.6)$$

Această ecuaţie o regăsim de asemenea implementată în figura 3.6.

Ultimele opt microcomenzi codificate în câmpul SHIFT & OTHER OPERATIONS sunt microcomenzi din categoria *„other”*. Microcomanda +2PC (care incrementează registrul PC cu 2) este urmată în tabelul 3.11 de şase microcomenzi care setează sau resetează bistabili de control din structura procesorului. Rolul acestor bistabili a fost explicat pe parcursul capitolului 2 (vezi instrucţiunile EI, DI, HALT şi faza de întrerupere). Ultima microcomandă (PdCOND) a fost codificată şi în microinstrucţiunea de transfer sincron. Această microcomandă setează *flag*-urile de condiţii din registrul FLAG în acord cu rezultatul obţinut pe ieşirea ALU.

Microinstrucţiunea de transfer asincron reprezintă un instrument pentru controlul operaţiilor cu memoria. Cu ajutorul acestei microinstruţiuni pot fi controlate următoarele tipuri de operaţii:

1) *fetch* instrucţiune din memorie. Se poate realiza cu microinstrucţiunea:

IFCH, +2PC

Această microinstrucţiune realizează operaţiile:

IR←MEM│$_{ADR}$;citirea codului instrucţiunii din memorie şi încărcarea ;acestuia în IR.

PC←PC+2 ;incrementarea registrului PC cu 2.

2) *fetch* operand din memorie. Se poate realiza cu microinstrucţiunea:

READ

Această microinstrucţiune realizează operaţia:

M←MEM│$_{ADR}$;citirea operandului din memorie şi încărcarea ;acestuia în registrul M.

3) *scriere în memorie*. De exemplu, microinstrucţiunea:

A SUM M, PdALU, WRITE realizează operaţia:

MEM│$_{ADR}$←A+M ;rezultatul operaţiei de adunare (A+M) este scris în ;memorie, în locaţia selectată de adresa existentă în ADR

4) *operaţii de tip „shift & other”*.

Exemple:

a) microinstrucţiunea DDA relizează deplasarea aritmetică la dreapta cu o poziţie a registrului A; va fi utilizată în microrutina de execuţie aferentă instrucţiunii ASR A.

b) microinstrucţiunea A(1)BVI realizează setarea *flag*-ului BVI (validarea întreruperilor); va fi utilizată în microrutina de execuţie aferentă instrucţiunii EI (*Enable Interrupt*).

Observaţii:

1. *Operaţiile „shift & other” pot fi codificate împreună (în aceeaşi microinstrucţiune) cu o operaţie de fetch (instrucţiune sau operand) sau de scriere în memorie. Fiind codificate în aceeaşi microinstrucţiune, devin operaţii concurente. Singura precauţie necesară se referă la evitarea conflictelor între operaţiile concurente.*

2. *Succesorul implicit al microinstrucţiunilor de transfer asincron este STEP. Fiind un succesor implicit, el nu a mai fost codificat în microinstrucţiune. Reamintim că acelaşi succesor implicit îl utilizăm şi la microinstrucţiunile de transfer sincron.*

3.5 Automatul SEQ

Automatul SEQ (***Seq**uencer*) generează comenzile interne unităţii de control microprogramate, comenzi prin care controlează *fetch*-ul şi execuţia microinstrucţiunilor.

3.5.1 Organigrama de funcţionare a automatului SEQ

Pentru a înţelege organigrama automatului SEQ (figura 3.7), interpretarea acesteia trebuie făcută în strânsă legătură cu schema bloc a unităţii de control microprogramate (figura 3.4) şi cu explicaţiile relative la această schemă bloc, (paragrafele 3.2 şi 3.4). După cum s-a arătat la 3.2, o microinstrucţiune este procesată principial în doi paşi (două perioade de tact succesive):

1. *fetch*-ul microinstrucţiunii (microinstrucţiunea selectată în MPM, de microadresa existentă în MAR, este citită şi încărcată în MIR). Pentru încărcarea în MIR, SEQ activează comanda *LdMIR* („*Load* MIR”).

2. execuţia microinstrucţiunii (decodificarea microinstrucţiunii din MIR şi activarea comenzilor codificate în microinstrucţiune). Pentru activarea comenzilor codificate în microinstrucţiune, SEQ validează, printr-o comandă „*Enable*” (EN1), decodificatoarele de microcomenzi.

Tot în pasul doi (în paralel cu execuţia microinstrucţiunii curente) se încarcă în MAR microadresa următoarei microinstrucţiuni. Pentru încărcarea acestei microadrese în MAR, SEQ activează comanda +1MAR („incrementare cu 1 MAR”) sau LdMAR („*Load* MAR”). În interiorul unei microrutine, SEQ trece secvenţial de la microinstrucţiunea curentă la cea următoare (activând comanda +1MAR). De pe sfârşitul unei microrutine, SEQ face salt pe începutul microrutinei următoare, cu ajutorul comenzii LdMAR care încarcă în MAR un câmp din microinstrucţiunea curentă (un câmp din MIR). În acest câmp de 12 biţi ($MIR_{11\div0}$) se codifică microadresa de salt.

Cele două operaţii succesive (*fetch* şi execuţie) sunt parcurse de către SEQ pentru fiecare microinstrucţiune din cadrul microprogramului executat. Pe de altă parte, la 3.4 am definit trei tipuri de microinstrucţiuni: de ramificaţie, de transfer sincron şi de transfer asincron. Drept urmare, SEQ trebuie să fie capabil să controleze execuţia acestor trei tipuri de microinstrucţiuni.

Din organigrama prezentată în figura 3.7 constatăm că automatul SEQ parcurge:

- 2 stări (starile 0 şi 1) în cazul în care controlează:
  - o microinstrucţiune de ramificaţie
  - o microinstrucţiune de transfer sincron
  - o microinstrucţiune de transfer asincron care are codificat NOP în câmpul MEMORY OPERATION (care nu activează o comandă efectivă de citire din sau de scriere în memorie şi care activează doar comenzi codificate în câmpul SHIFT & OTHER OPERATIONS).

- 4 stări (stările 0, 1, 2 şi 3) în cazul în care controlează o microinstrucţiune de transfer asincron care codifică altceva decât NOP în câmpul MEMORY OPERATION (care iniţiază efectiv o operaţie de citire din sau de scriere în memorie printr-o comandă IFCH, READ sau WRITE). În acest caz, în stările 2 şi 3, SEQ controlează operaţia de citire din sau de scriere în memorie.

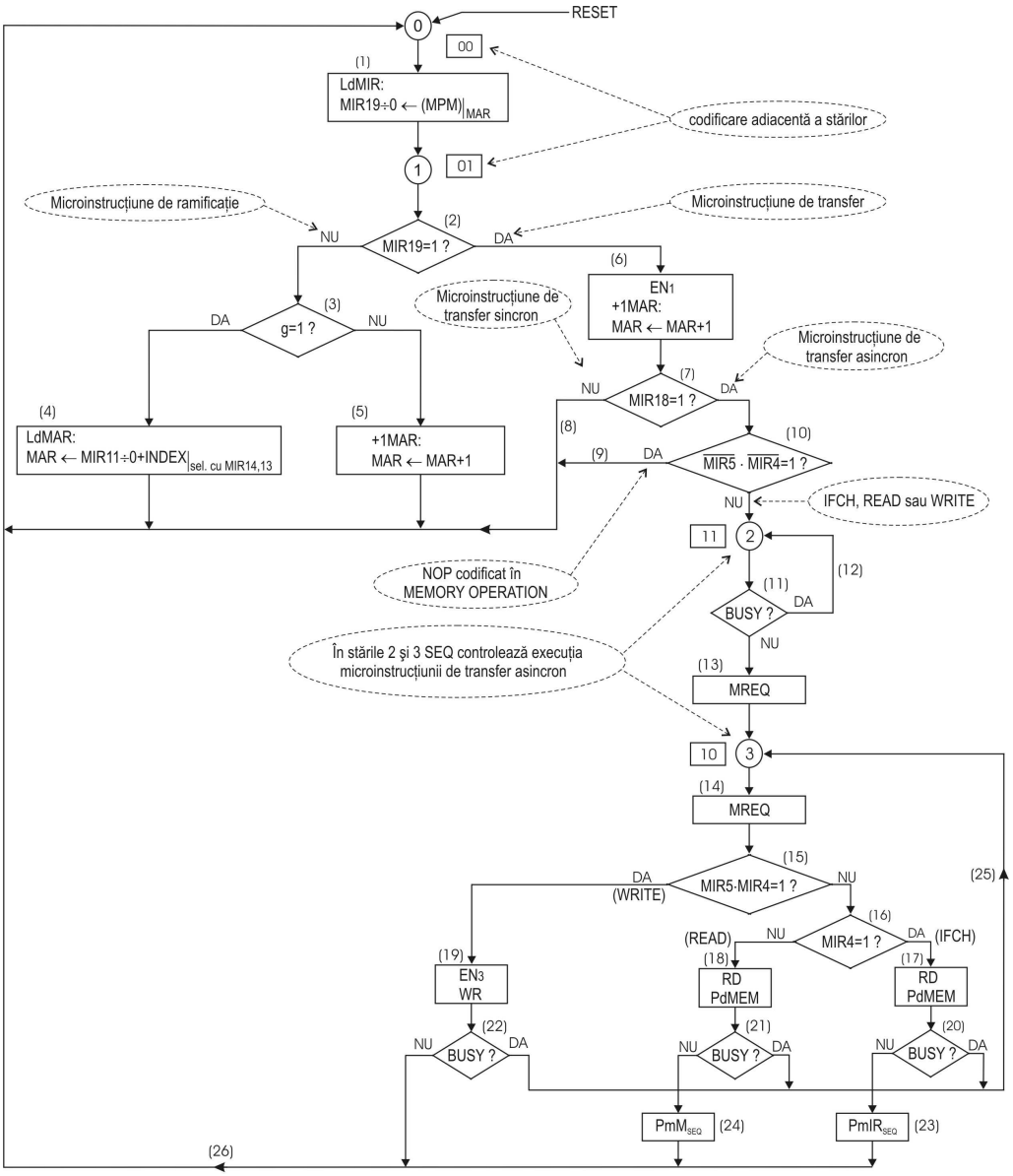


Fig 3.7 Organigrama de funcţionare a automatului SEQ

În starea 0, automatul SEQ activează comanda LdMIR (blocul **(1)** din figura 3.7), prin care încarcă în MIR microinstrucţiunea selectată în MPM de adresa existentă în MAR (*fetch* –ul microinstrucţiunii).

În starea 1, SEQ generează comenzi dependente de codul microinstrucţiunii încărcate în MIR în starea anterioară:

a). Dacă în MIR se află o microinstrucţiune de ramificaţie (ieşirea NU din blocul de decizie **(2)**) şi funcţia globală de ramificaţie este g=0 (ieşirea NU din blocul de decizie **(3)**) avem de-a face cu o ramificaţie condiţionată cu condiţia testată neândeplinită. SEQ va genera comanda +1MAR **(5)** prin care registrul MAR este incrementat (MAR←MAR+1). Această comandă (operaţie) va determina trecerea, în ordine secvenţială, la următoarea microinstrucţiune (deoarece condiţia de salt este neândeplinită). Altfel spus, se execută succesorul STEP.

b). Dacă în MIR se află o microinstrucţiune de ramificaţie **(2)** şi g=1 **(3)** (ramificaţie necondiţionată sau condiţionată cu condiţia testată îndeplinită), SEQ va genera comanda: LdMAR **(4)**. Prin urmare, se va face un salt în microcod. Microadresa de salt se obţine adunând INDEX–ul selectat de câmpul $MIR_{14\div13}$ la microadresa de bază conţinută în câmpul $MIR_{11\div0}$ al microinstrucţiunii curente. Altfel spus, se execută operaţia:

$$MAR \leftarrow MIR_{11\div0} + INDEX\big|_{\text{selectat de } MIR_{14\div13}}$$

În această manieră se execută succesorii JUMPI din tabelul 3.2.

Dacă $MIR_{14,13}=00_2$ atunci INDEX–ul selectat va avea valoarea zero (vezi tabelul 3.3). În aceste condiţii se va executa operaţia:

$$MAR \leftarrow MIR_{11\div0}$$

Se va face deci saltul la microinstrucţiunea a cărei adresă este conţinută în câmpul $MIR_{11\div0}$ al microinstrucţiunii curente. În această manieră se execută succesorii JUMP din tabelul 3.1. În concluzie, dacă indexul selectat de $MIR_{14\div13}$ are valoarea zero, sucesorii JUMPI din tabelul 3.2 se transformă în succesorii JUMP echivalenţi din tabelul 3.1.

În toate cazurile descrise până aici, din starea 1, SEQ revine în starea 0 (reia secvenţa de stări pentru procesarea următoarei microinstrucţiuni).

c). Dacă în MIR se află o microinstrucţiune de transfer (sincron sau asincron), SEQ va genera comenzile EN_1 şi +1MAR **(6)**. Prin comanda EN_1 va valida decodificatoarele de microcomenzi (fig 3.4) asigurând astfel activarea microcomenzilor codificate în microinstrucţiune. Reamintim faptul că, spre deosebire de microinstrucţiunea de ramificaţie, microinstrucţiunile de transfer au câmpuri în care sunt codificate microcomenzi. Aceste microcomenzi trebuie activate în starea 1 a automatului SEQ (starea de execuţie a microinstrucţiunii). Tot în starea 1 a automatului SEQ trebuie încărcată în MAR microadresa următoarei

microinstrucţiuni. Această încărcare este realizată de comanda +1MAR. În aceste condiţii se va executa operaţia de incrementare a registrului MAR:

MAR←MAR+1

Se observă de aici că succesorul **implicit** al microinstrucţiunilor de transfer este STEP (în acord cu cele stipulate la 3.4.2).

În continuare, evoluţia autmatului SEQ depinde de tipul transferului codificat în microinstrucţiunea curentă:

c1). Dacă microinstrucţiunea curentă este o microinstrucţiune de transfer sincron **(8)**, SEQ revine în starea 0 (reia secvenţa de stări pentru microinstrucţiunea următoare).

c2). Dacă microinstrucţiunea curentă este o microinstrucţiune de transfer asincron dar care are codificat NOP în câmpul MEMORY OPERATION **(9)**, SEQ revine în starea 0 (reia secvenţa de stări pentru microinstrucţiunea următoare).

c3). Dacă microinstrucţiunea curentă este o microinstrucţiune de transfer asincron **efectivă**, care are codificat IFCH, READ sau WRITE în câmpul MEMORY OPERATION (ieşirea NU din blocul de decizie **(10)**), SEQ trece în starea 2. În continuare (în stările 2 şi 3) SEQ va controla operaţia de transfer procesor–memorie, care poate fi:

IFCH – citire cod instrucţiune din memorie

READ – citire date (operand) din memorie

WRITE – scriere date (rezultat) în memorie.

În starea 2 este testat semnalul BUSY **(11)**; BUSY este semnalul de stare aferent memoriei. Dacă BUSY=1 (memoria este ocupată cu un transfer de date anterior declanşat sau cu un ciclu de regenerare), SEQ rămâne în starea 2 **(12)** aşteptând dezactivarea semnalului BUSY. Când BUSY "cade" în 0 (când memoria devine liberă) automatul SEQ activează comanda MREQ (*Memory REQuest* = cerere de acces la memorie) **(13)** şi 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 menţine cererea MREQ activă **(14)** şi va controla efectiv operaţia de transfer procesor–memorie (codificată în câmpul MEMORY OPERATION al microinstrucţiunii):

c3.1) Dacă în câmpul MEMORY OPERATION este codificată microcomanda IFCH (ieşirea DA din blocul de decizie **(16)**), SEQ menţine activă comanda MREQ, activează suplimentar comenzile RD şi PdMEM **(17)** şi testează BUSY **(20)** (aşteaptă încheierea acestui ciclu de citire din memorie **(25)**). Când BUSY "cade" în 0 (când ciclul se încheie), SEQ menţine active comenzile MREQ, RD

şi PdMEM şi activează suplimentar comanda $PmIR_{SEQ}$ **(23),** încărcând astfel informaţia citită din memorie (codul instrucţiunii) în registrul IR. Revine apoi în starea 0 **(26)** pentru procesarea următoarei microinstrucţiuni. Să notăm că RD reprezintă chiar comanda de citire pe care procesorul o emite, la nivel *hardware*, către memorie (fig 2.2).

c3.2) Dacă în câmpul MEMORY OPERATION este codificată microcomanda READ (ieşirea NU din blocul de decizie **(16)**), protocolul decurge analog, cu deosebirea că, în locul comenzii $PmIR_{SEQ}$, SEQ va activa comanda PmM_{SEQ} (**(18) (21) (24)**), deoarece în acest caz, informaţia citită din memorie nu reprezintă un cod de instrucţiune ci un operand şi ca atare acesta trebuie încărcat în registrul M. Să observăm că ambele microcomenzi (IFCH şi READ) fac citire din memorie; cele două microcomenzi diferă doar prin destinaţia informaţiei citite din memorie: IFCH încarcă informaţia în registrul IR iar READ în registrul M. Se va vedea la "Microprogramul de Emulare" că IFCH va fi utilizată în microrutina "***Fetch* Instrucţiune**" iar READ în microrutinele "***Fetch* Operand**".

Observaţie*: Am utilizat indicele „SEQ" pentru comenzile de încărcare paralelă în registrele IR şi respectiv M ($PmIR_{SEQ}$, PmM_{SEQ}) deoarece aceste comenzi sunt elaborate direct de către automatul SEQ, în starea 3, dacă microinstrucţiunea curentă este una de transfer asincron ($PmIR_{SEQ}$ este activată dacă în microinstrucţiunea curentă este codificată microcomanda IFCH iar PmM_{SEQ} este activată dacă în microinstrucţiunea curentă este codificată microcomanda READ). Comenzile PmIR şi respectiv PmM au fost codificate şi în microinstrucţiunea de transfer sincron (câmpul „DESTINAŢIE RBUS din figura 3.3). Aceste microcomenzi au fost prevăzute pentru transferurile registru-registru (transferuri interne procesorului). În aceast caz, comenzile PmIR şi respectiv PmM vor fi activate în starea 1 (starea de execuţie a microinstrucţiunii). Pentru diferenţiere, comenzile PmIR şi respectiv PmM codificate în microinstrucţiunea de transfer sincron, ar trebui marcate cu indicele „S" ($PmIR_S$, PmM_S). Comenzile efective (comenzile globale) de încărcare paralelă în registrele IR şi respectiv M se vor obţine printr-o funcţie SAU:*

$$\begin{aligned} PmIR &= PmIR_{SEQ} + PmIR_S \\ PmM &= PmM_{SEQ} + PmM_S \end{aligned} \qquad (3.7)$$

Pentru împlementare în hardware, ecuaţiile (3.7) revendică două porţi SAU.

c3.3) Dacă în câmpul MEMORY OPERATION este codificată microcomanda WRITE (ieşirea DA din blocul de decizie **(15)**), SEQ urmează să controleze o scriere în memorie şi drept urmare, activează comenzile EN_3 şi WR **(19)**. Prin comanda WR declanşează operaţia de scriere în memorie (fig 2.2) iar prin testarea semnalului BUSY **(22)** aşteaptă încheierea acestei scrieri.

Reamintim faptul că, la scriere, procesorul trebuie să genereze şi datele de scris în memorie. Generarea acestor date se face cu ajutorul unor microcomenzi codificate în câmpurile SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS, câmpuri care au semnificaţie numai dacă în câmpul MEMORY OPERATION este codificată microcomanda WRITE (vezi 3.4.2.B). Comanda EN_3 va valida decodificatoarele de microcomenzi aferente celor patru câmpuri. Devine evident faptul că vor fi scrise în memorie datele prezente pe RBUS pe durata stării 3 (cu PdMEM=0, prezente şi pe MBUS; vezi fig 2.36). Emisia pe RBUS a datelor de scris în memorie este asigurată deci de microcomenzile codificate în cele patru câmpuri (SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS).

Când operaţia de scriere în memorie se încheie (când semnalul BUSY se dezactivează), SEQ revine în starea 0 pentru procesarea următoarei microinstrucţiuni **(26)**.

Observaţie: *SEQ ar putea să nu aştepte încheierea ciclului de scriere şi să revină imediat în starea 0 (eficienţă!). Chiar dacă pe microinstrucţiunea următoare s-ar declanşa un nou acces la memorie, nu vor apărea probleme deoarece, pe următoarea microinstrucţiune, SEQ va aştepta în starea 2 încheierea ciclului de scriere în memorie anterior lansat. Pentru a implementa însă acest protocol (mai eficient pe operaţiile de scriere) este necesar, la nivelul memoriei, un* ***registru buffer de date*** *care să reţină datele de scris; când SEQ iese din starea 3, datele de scris dispar de pe RBUS şi implicit de pe MBUS (figura 3.7).*

3.5.2 Graful de tranziţii al automatului SEQ

Graful de tranziţii al automatului SEQ se elaborează pe baza organigramei de funcţionare din figura 3.7. Acest graf este redat în figura 3.8.

În graful de tranziţii apar cele patru stări din organigramă, cu tranziţiile dintre stări (figurate cu ajutorul unor arce orientate). Am consemnat în figura 3.8 şi modul de codificare a stărilor; cele patru stări pot fi codificate cu ajutorul a două semnale de stare. Subliniem faptul că stările au fost codificate adiacent (două stări interconectate printr-un arc de tranziţie diferă între ele printr-un singur bit de stare). Regula codificării adiacente a fost respectată pe toate arcele din graf deoarece, la tranziţia dintre două stări codificate neadiacent, se pot genera impulsuri scurte pe ieşirile automatului (comenzi parazite). Codificarea adiacentă elimină comenzile parazite.

Pe arcele care reprezintă tranziţii condiţionate am trecut şi condiţiile care validează tranziţiile respective.

Exemple:

1) Tranziţia din starea 2 în starea 3 se poate face numai dacă BUSY=0 (memoria nu este ocupată cu un ciclu de citire sau scriere anterior startat sau cu un ciclu de regenerare). Această condiţie rezultă evident din organigrama automatului SEQ (figura 3.7).

2) Revenirea (tranziţia) din starea 1 în starea 0 se realizează numai dacă este îndeplinită una din următoarele trei condiţii:

   a) micoinstrucţiunea curentă (microinstrucţiunea din MIR) este o microinstrucţiune de ramificaţie.

   b) micoinstrucţiunea curentă este o microinstrucţiune de transfer sincron.

   c) micoinstrucţiunea curentă este o microinstrucţiune de transfer asincron care are codificat NOP în câmpul MEMORY OPERATION (care nu activează o comandă efectivă de citire din sau de scriere în memorie).

Cele trei condiţii sunt sintetizate în ecuaţia (3.8).

$$\overline{MIR_{19}} + MIR_{19} \bullet \overline{MIR_{18}} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR_5} \bullet \overline{MIR_4} = 1 \qquad (3.8)$$

Ecuaţia (3.8) conţine o reuniune (funcţie SAU) de trei produse logice. În primul termen ($\overline{MIR_{19}}$) este codificată condiţia 2.a), în al doilea produs ($MIR_{19} \bullet \overline{MIR_{18}}$) este codificată condiţia 2.b) iar în ultimul produs ($MIR_{19} \bullet MIR_{18} \bullet \overline{MIR_5} \bullet \overline{MIR_4}$) este codificată condiţia 2.c).

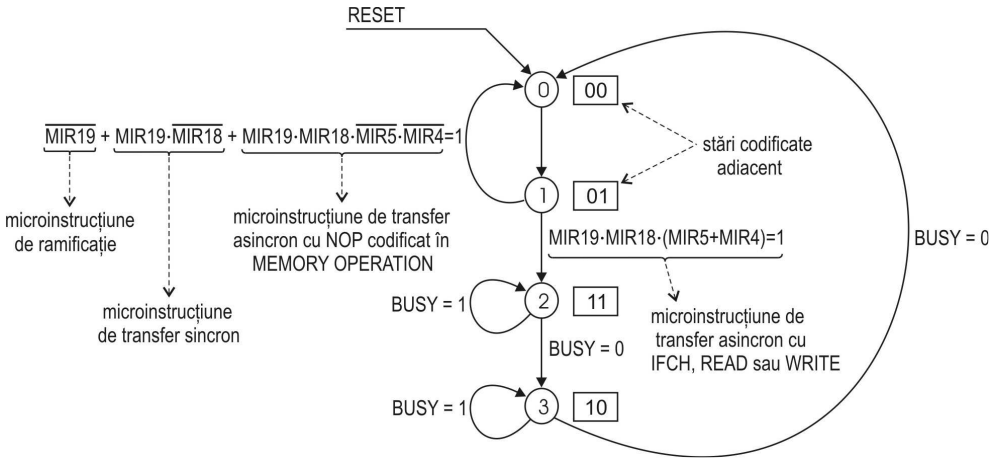
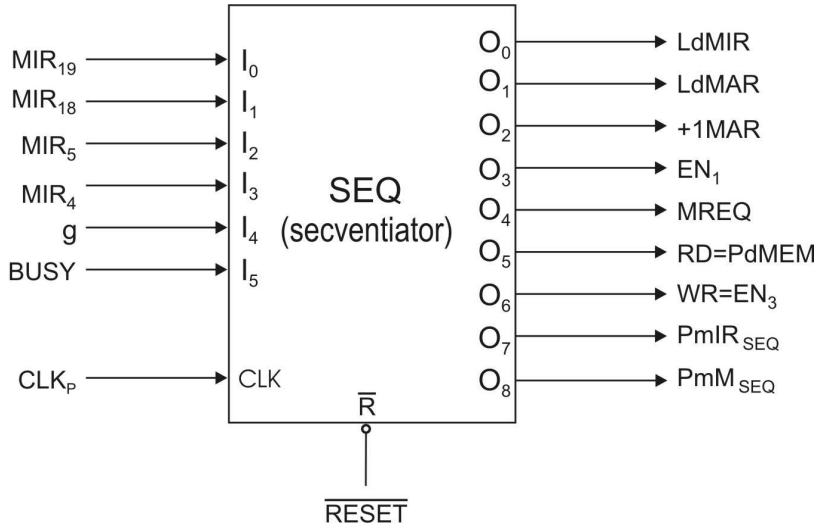


Fig 3.8 Graful de tranziţii al automatului SEQ

Să menţionăm că organigrama (figura 3.7) şi graful de tranziţii (figura 3.8) sunt necesare în etapa de proiectare a automatului SEQ.

3.5.3 Schema bloc a automatului SEQ

Schema bloc a automatului SEQ se obţine pornind de la organigrama automatului din figura 3.7. Variabilele testate în organigramă vor fi intrări în automat, iar comenzile activate în organigramă (în diferitele stări) vor fi ieşiri din automat. Automatul SEQ este un automat secvenţial sincron; trecerea din starea curentă în starea următoare se face sub controlul semnalului de tact al procesorului CLK_P (figura 3.1). Iniţializarea automatului se face cu ajutorul semnalului $\overline{RESET}$ care realizează de fapt iniţializarea procesorului şi respectiv calculatorului (în ansamblu). Schema bloc a automatului SEQ este redată în figura 3.9.



I_i -intrări în automat (*inputs*)
O_i -ieşiri din automat (*outputs*)
FIG 3.9 Schema bloc a automatului SEQ

În organigrama automatului din figura 3.7 pot fi identificate şi comenzi identice: comanda PdMEM este identică cu comanda RD iar comanda EN_3 este identică cu comanda WR. Pe această bază putem reduce numărul de ieşiri din automat (simplificări în *hardware*).

3.5.4 Implementarea automatului SEQ cu memorie ROM şi registru

În varianta de impementare cu ROM (***R**ead **O**nly **M**emory*) şi registru, automatul conţine un registru şi o memorie ROM (figura 3.10). În fond, schema din figura 3.10 reprezintă o detaliere a schemei bloc din figura 3.9. Intrările în automat şi respectiv ieşirile din automat,

în ambele reprezentări, nu pot fi decât identice. Intrările şi respectiv ieşirile rezultă, după cum am mai spus, din organigrama automatului (figura 3.7).

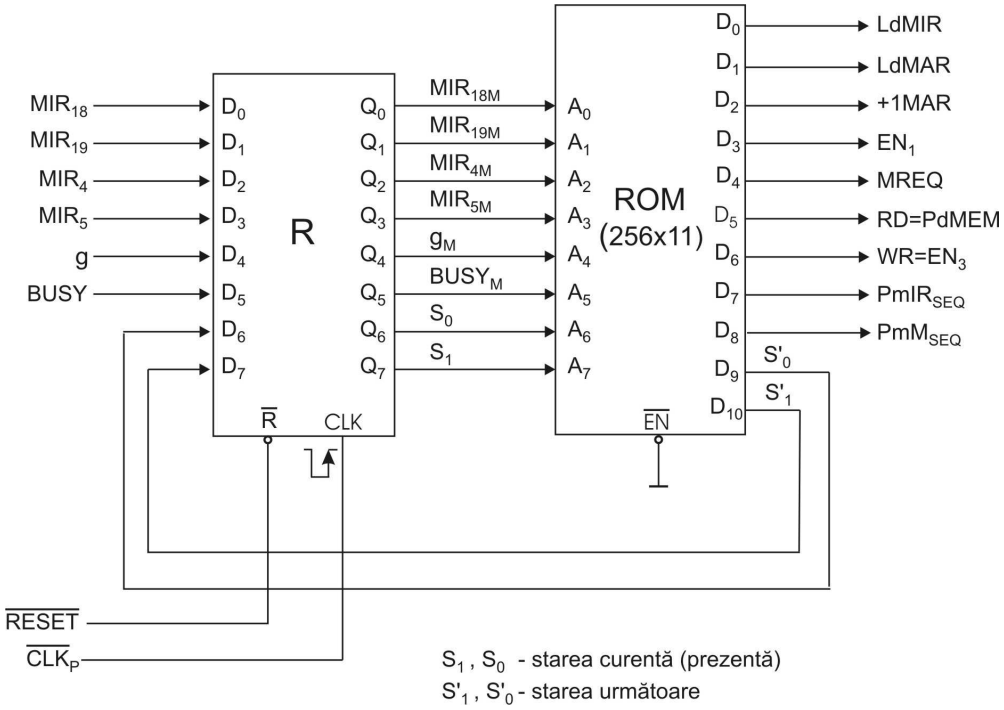


Fig 3.10 Implementarea automatului SEQ cu ROM şi registru

În schema detaliată din figura 3.10 apar suplimentar:

- semnalele de legătură dintre registru si ROM. Aceste semnale reprezintă ieşirile de date din registru şi respectiv intrările de adresă în ROM. În fond, frontul activ al fiecărui impuls de tact ($\overline{CLK_P}$) încarcă în registrul R semnalele aplicate pe intrările de date ale registrului: MIR_{19}, MIR_{18}, MIR_5, MIR_4, g şi BUSY. Aceste semnale reprezintă intrările în automat (variabilele testate în organigrama de funcţionare a automatului). Prin încărcare, aceste semnale de intrare devin „intrări memorate”. Intrările memorate sunt de fapt semnalele de legătură dintre registru şi ROM; din acest motiv am adnotat semnalele de legătură cu indicele M. Să exemplificăm:

    1. variabila MIR_{19} de pe intrarea automatului, după memorarea în registru, devine MIR_{19M} (MIR_{19} memorat)

    2. funcţia globală de ramificaţie (*g*) de la intrare, după memorare , devine g_M (*g* memorat)

- cele două semnale de stare aferente automatului. Semnalele S_1 şi respectiv S_0 reprezintă starea prezentă (curentă), iar $S_1^{'}$ şi $S_0^{'}$ reprezintă starea următoare a automatului. După încărcarea în registru realizată de următorul impuls $\overline{CLK_P}$, starea următoare ($S_1^{'}, S_0^{'}$), va deveni noua stare prezentă (S_1,S_0).

Organigrama automatului SEQ conţine 4 stări pe care le-am codificat cu ajutorul celor două semnale de stare, S_1 şi respectiv S_0. Codificarea (adiacentă) a stărilor este redată în tabelul 3.12

S_1	S_0	Stare
0	0	ST_0 (starea 0)
0	1	ST_1 (starea 1)
1	1	ST_2 (starea 2)
1	0	ST_3 (starea 3)

Tabelul 3.12 Codificarea stărilor automatului SEQ

Starea iniţială (S_1,S_0=0,0) se instalează prin activarea comenzii $\overline{RESET}$ care resetează registrul R. După eliberarea registrului din *reset*, automatul va evolua sincron cu semnalul de tact $\overline{CLK_P}$.

Resetat, registrul R va aplica o adresă egală cu zero pe intrările de adresă ale ROM-ului, ceea ce înseamnă că automatul SEQ se află în starea ST_0 (S_1,S_0=0,0). Conform organigramei automatului (figura 3.7), în starea ST_0 trebuie activată comanda LdMIR şi din starea ST_0 automatul trebuie să treacă în starea ST_1. Activarea comenzii LdMIR, şi respectiv trecerea în starea ST_1 sunt necondiţionate (nu sunt condiţionate de nici-o variabilă de intrare). Asta înseamnă că, în starea ST_0, indiferent de valorile variabilelor de intrare, pe ieşirile de date ale ROM-ului trebuie să avem:

- comanda LdMIR activă (LdMIR=1)
- starea următoare $S_1^{'}, S_0^{'}$=0,1
- toate celelalte ieşiri trebuie să fie inactive.

Prin urmare, pe baza organigramei automatului SEQ (figura 3.7), relativ la programarea ROM-ului (tabelul 3.13), putem concluziona că, la adresele $A_{7\div0}$=00XXXXXX$_2$, trebuie „arsă” (programată) informaţia $D_{10\div0}$=01000000001$_2$ (starea următoare $S_1^{'}, S_0^{'}$=0,1 şi LdMIR=1; celelalte ieşiri trebuie să rămână inactive în starea ST_0). Ultimii şase biţi de adresă ($A_{7\div0}$=00XXXXXX$_2$) au fost marcaţi cu X deoarece valorile lor nu contează (pot fi

zero sau unu). Această concluzie rezulta din organigrama SEQ: **comanda LdMIR, precum şi tranziţia din starea ST_0 în starea ST_1, nu sunt condiţionate de nici o variabilă de intrare în automat**. În această manieră am obţinut primul rând din tabelul 3.13, tabel care sintetizează conţinutul ROM-ului din structura automatului SEQ. În această primă linie din tabel am sintetizat de altfel conţinutul primului sfert din matricea de memorie ROM.

În continuare, conţinutul matricii ROM trebuie elaborat pe baza succesiunilor de stări din organigrama SEQ. Primul impuls de tact ($\overline{CLK_P}$) care survine după eliberarea automatului din RESET (mai precis, frontul urcător al primului impuls $\overline{CLK_P}$), va încărca în registrul R:

- valorile momentane ale variabilelor de intrare: MIR_{19}, MIR_{18}, MIR_5, MIR_4, g, şi BUSY.

- starea următoare ($S_1^{'}, S_0^{'}$=0,1), care prin încărcarea în registru devine noua stare prezentă (S_1,S_0=0,1).

Prin această încărcare în registru, variabilele de intrare devin variabile memorate iar starea următoare ($S_1^{'}, S_0^{'}$=0,1) devine stare prezentă (S_1,S_0=0,1). Conform organigramei SEQ, în starea ST_1 sunt testate următoarele variabile de intrare: MIR_{19}, MIR_{18}, MIR_5, MIR_4 şi g. Sunt posibile următoarele cazuri:

- MIR_{19}=0, g=0, MIR_{18}=X, MIR_5=X, MIR_4=X (microinstrucţiune de ramificaţie cu condiţie testată neândeplinită). În acesst caz SEQ trebuie să activeze comanda +1MAR (blocul **(5)** din organigramă) şi să treacă în starea ST_0. Rezultă că la adresele $A_{7\div0}$=01X0XX0X_2 trebuie „arsă” informaţia $D_{10\div0}$=00000000100$_2$ (starea următoare $S_1^{'}, S_0^{'}$=0,0 şi +1MAR=1). Am obţinut astfel al doilea rând din tabelul 3.13.

- MIR_{19}=0, g=1, MIR_{18}=X, MIR_5=X, MIR_4=X (microinstrucţiune de ramificaţie cu condiţie testată îndeplinită). În acesst caz SEQ trebuie să activeze comanda LdMAR (blocul **(4)** din organigramă) şi să treacă în starea ST_0. Rezultă că la adresele $A_{7\div0}$=01X1XX0X_2 trebuie „arsă” informaţia $D_{10\div0}$=00000000010$_2$ (starea următoare $S_1^{'}, S_0^{'}$=0,0 şi LdMAR=1). Am obţinut astfel al treilea rând din tabelul 3.13.

- MIR_{19}=1, g=X, MIR_{18}=0, MIR_5=X, MIR_4=X (microinstrucţiune de transfer sincron). În acest caz SEQ trebuie să activeze comenzile EN_1 şi +1MAR (blocul **(6)** din organigramă) şi să treacă în starea ST_0. Rezultă că la adresele $A_{7\div0}$=01XXXX10$_2$ trebuie „arsă” informaţia $D_{10\div0}$=00000001100$_2$ (starea următoare $S_1^{'}, S_0^{'}$=0,0 şi comenzile: EN_1=1 şi +1MAR=1). Am obţinut astfel al patrulea rând din tabelul 3.13.

- MIR_{19}=1, g=X, MIR_{18}=1, MIR_5=0, MIR_4=0 (microinstrucţiune de transfer asincron cu microcomanda NOP codificată în câmpul MEMORY OPERATION). În acesst caz SEQ trebuie să activeze comenzile EN_1 şi +1MAR (blocul **(6)** din organigramă) şi să treacă în starea ST_0. Rezultă că la adresele $A_{7\div0}=01XX0011_2$ trebuie „arsă" informaţia $D_{10\div0}=00000001100_2$ (starea următoare $S_1^{'}, S_0^{'}$=0,0 şi comenzile: EN_1=1 şi +1MAR=1). Am obţinut astfel al cincilea rând din tabelul 3.13.

- MIR_{19}=1, g=X, MIR_{18}=1, MIR_5=0, MIR_4=1 (microinstrucţiune de transfer asincron cu microcomanda IFCH codificată în câmpul MEMORY OPERATION). Ca şi în cazul anterior, SEQ trebuie să activeze comenzile EN_1 şi +1MAR (blocul **(6)** din organigramă) şi să treacă în starea ST_2. Rezultă că la adresele $A_{7\div0}=01XX0111_2$ trebuie „arsă" informaţia $D_{10\div0}=11000001100_2$ (starea următoare $S_1^{'}, S_0^{'}$=1,1 şi comenzile: EN_1=1 şi +1MAR=1). Am obţinut astfel al şasetelea rând din tabelul 3.13.

- MIR_{19}=1, g=X, MIR_{18}=1, MIR_5=1, MIR_4=0 (microinstrucţiune de transfer asincron cu microcomanda READ codificată în câmpul MEMORY OPERATION). În acesst caz SEQ trebuie să activeze comenzile EN_1 şi +1MAR (blocul **(6)** din organigramă) şi să treacă în starea ST_2. Rezultă că la adresele $A_{7\div0}=01XX1011_2$ trebuie „arsă" informaţia $D_{10\div0}=11000001100_2$ (starea următoare $S_1^{'}, S_0^{'}$=1,1 şi comenzile: EN_1=1 şi +1MAR=1). Am obţinut astfel al şaptelea rând din tabelul 3.13.

- MIR_{19}=1, g=X, MIR_{18}=1, MIR_5=1, MIR_4=1 (microinstrucţiune de transfer asincron cu microcomanda WRITE codificată în câmpul MEMORY OPERATION). Ca în cazul anterior, SEQ trebuie să activeze comenzile EN_1 şi +1MAR (blocul **(6)** din organigramă) şi să treacă în starea ST_2. Rezultă că la adresele $A_{7\div0}=01XX1111_2$ trebuie „arsă" informaţia $D_{10\div0}=11000001100_2$ (starea următoare $S_1^{'}, S_0^{'}$=1,1 şi comenzile: EN_1=1 şi +1MAR=1). Am obţinut astfel al optulea rând din tabelul 3.13.

În starea ST_2 este testată doar variabila BUSY (blocul **(11)** din orbanigramă. Sunt posibile 2 cazuri:

- BUSY=1 (memoria ocupată cu un ciclu lansat de un alt dispozitiv, de exemplu un dispozitiv DMA, sau cu un ciclu de regenerare). În acest caz, SEQ trebuie să aştepte eliberarea memoriei (bucla **(12)** din organigramă).Rezultă că la adresele $A_{7\div0}=111XXXXX_2$ trebuie „arsă" informaţia $D_{10\div0}=11000000000_2$ (starea următoare $S_1^{'}, S_0^{'}$=1,1 şi nici-o comandă activă). Am obţinut astfel al nouălea rând din tabelul 3.13.

- BUSY=0 (memoria liberă). În acest caz, SEQ activează cererea de acces la memorie MREQ (blocul **(13)** din organigramă) şi trece în starea ST_3. Rezultă că la adresele $A_{7\div0}=110XXXXX_2$ trebuie „arsă” informaţia $D_{10\div0}=10000010000_2$ (starea următoare $S_1^{'}, S_0^{'}=1,0$ şi comanda MREQ activă). Am obţinut astfel al zecelea rând din tabelul 3.13.

În starea ST_3, SEQ controlează operaţia de transfer proceor-memorie. Menţine activă cererea MREQ (activată încă din starea anterioară!) şi testează MIR_5 şi MIR_4 pentru a stabili ce operaţie va efectua (***fetch* instrucţiune**, ***fetch* operand** sau **scriere**). În toate cele 3 cazurile, testează BUSY (blocurile **(20)**, **(21)** şi **(22)**) pentru a aştepta încheierea transferului procesor-memorie (bucla **(25)** din organigramă). Când ciclul de transfer (***fetch* instrucţiune**, ***fetch* operand** sau **scriere**) se încheie, SEQ trece în starea ST_0 (pentru *fetch*-ul următoarei microinstrucţiuni). Prin urmare:

- Dacă $MIR_5=0$, $MIR_4=1$ şi BUSY=1 (*fetch* instrucţiune), SEQ activează comenzile MREQ şi RD=PdMEM (blocul **(17)** din organigramă) şi ciclează în starea ST_3 (până la încheierea operaţiei de citire din memorie). Prin urmare, la adresa $A_{7\div0}=101X01XX_2$ trebuie „arsă” informaţia $D_{10\div0}=10000110000_2$. (rândul 11 din tabelul 3.13).

  Când BUSY se dezactivează (când ciclul de citire din memorie se încheie), SEQ activează suplimentar $PmIR_{SEQ}$ (încărcă în IR codul instrucţiunii extrase din memorie) şi revine în starea ST_0. Prin urmare, la adresa $A_{7\div0}=100X01XX_2$ trebuie „arsă” informaţia $D_{10\div0}=00010110000_2$. (rândul 14 din tabelul 3.13).

- Dacă $MIR_5=1$, $MIR_4=0$ şi BUSY=1 (*fetch* operand), SEQ activează comenzile MREQ şi RD=PdMEM (blocul **(18)** din organigramă) şi ciclează în starea ST_3 (până la încheierea operaţiei de citire din memorie). Prin urmare, la adresa $A_{7\div0}=101X10XX_2$ trebuie „arsă” informaţia $D_{10\div0}=10000110000_2$. (rândul 12 din tabelul 3.13).

  Când BUSY se dezactivează (când ciclul de citire din memorie se încheie), SEQ activează suplimentar PmM_{SEQ} (încărcă în M operandul extras din memorie) şi revine în starea ST_0. Prin urmare, la adresa $A_{7\div0}=100X10XX_2$ trebuie „arsă” informaţia $D_{10\div0}=00100110000_2$. (rândul 15 din tabelul 3.13).

- Dacă $MIR_5=1$, $MIR_4=1$ şi BUSY=1 (scriere), SEQ activează comenzile MREQ şi $WR=EN_3$ (blocul **(19)** din organigramă) şi ciclează în starea ST_3 (până la încheierea operaţiei de scriere în memorie). Prin urmare, la adresa $A_{7\div0}=101X11XX_2$ trebuie „arsă” informaţia $D_{10\div0}=10001010000_2$. (rândul 13 din tabelul 3.13).

  Când BUSY se dezactivează (când ciclul de scriere în memorie se încheie), SEQ revine în starea ST_0. Prin urmare, la adresa $A_{7\div0}=100X11XX_2$ trebuie „arsă” informaţia $D_{10\div0}=00001010000_2$. (ultimul rând din tabelul 3.13).

Proiectarea automatului SEQ se încheie cu stabilirea conţinutului memoriei ROM.

Tipurile de microinstrucţiuni tratate	Intrări SEQ (intrări de adresă în PROM)								Ieşiri SEQ (ieşiri de date din PROM)										
	S_1	S_0	BUSY	g	MIR_5	MIR_4	MIR_{19}	MIR_{18}	S_1'	S_0'	PmM_{SEQ}	$PmIR_{SEQ}$	WR EN_3	RD PdMEM	MREQ	EN_1	+IMAR	LdMAR	LdMIR
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Toate tipurile	0	0	x	x	x	x	x	x	0	1	0	0	0	0	0	0	0	0	1
Ramificaţie	0	1	x	0	x	x	0	x	0	0	0	0	0	0	0	0	1	0	0
	0	1	x	1	x	x	0	x	0	0	0	0	0	0	0	0	0	1	0
Transfer sincron	0	1	x	x	x	x	1	0	0	0	0	0	0	0	0	1	1	0	0
Transfer asincron	0	1	x	x	0	0	1	1	0	0	0	0	0	0	0	1	1	0	0
	0	1	x	x	0	1	1	1	1	1	0	0	0	0	0	1	1	0	0
	0	1	x	x	1	0	1	1	1	1	0	0	0	0	0	1	1	0	0
	0	1	x	x	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0
Transfer asincron	1	1	1	x	x	x	x	x	1	1	0	0	0	0	0	0	0	0	0
	1	1	0	x	x	x	x	x	1	0	0	0	0	0	1	0	0	0	0
Transfer asincron	1	0	1	x	0	1	x	x	1	0	0	0	0	1	1	0	0	0	0
	1	0	1	x	1	0	x	x	1	0	0	0	0	1	1	0	0	0	0
	1	0	1	x	1	1	x	x	1	0	0	0	1	0	1	0	0	0	0
	1	0	0	x	0	1	x	x	0	0	0	1	0	1	1	0	0	0	0
	1	0	0	x	1	0	x	x	0	0	1	0	0	1	1	0	0	0	0
	1	0	0	x	1	1	x	x	0	0	0	0	1	0	1	0	0	0	0

Tabelul 3.13 Conţinutul memoriei ROM din componenţa automatului SEQ

3.5.5 Evaluarea funcţională a automatului SEQ

După definitivarea proiectării, se poate face evaluarea funcţională a automatului SEQ, prin simularea anumitor scenarii. După cum am stabilit în faza de proiectare, automatul SEQ controlează procesarea (fazele de *fetch* şi respectiv execuţie) a trei tipuri de microinstrucţiuni: microinstrucţiunea de ramificaţie, de transfer sincron şi de transfer asincron. Cea mai complexă sarcină pentru SEQ apare la microinstrucţiunea de transfer asincron care lansează efectiv o operaţie de transfer procesor-memorie (care conţine o microcomandă IFCH, READ sau WRITE codificată în câmpul MEMORY OPERATION). Pentru procesarea unei asemenea microinstrucţiuni SEQ parcurge toate cele patru stări ($ST_0 \div ST_3$). Memoria este de regulă mai lentă decât procesorul (ciclul memoriei este mai lung decât perioada de tact procesor). Prin urmare, operaţiile de citire din sau de scriere în memorie, pe care SEQ le controlează în starea ST_3, se vor desfăşura pe durata mai multor perioade de tact procesor (perioade de tact SEQ). Operaţiile cu memoria sunt controlate printr-un protocol asincron bazat pe semnale de *hand-shaking*. Cele două semnale de *hand-shaking* utilizate sunt: MREQ –cererea de acces la memorie activată de către procesor (de către SEQ) şi respectiv BUSY –semnalul de stare aferent memoriei (semnalul de răspuns la cererea MREQ). Acest protocol este implementat în starea ST_3.

Cronograma din figura 3.11 simulează funcţionalitatea automatului SEQ, atunci când acesta controlează procesarea unei microinstrucţiuni de transfer asincron cu microcomanda READ codificată în câmpul MEMORY OPERATION. Prin urmare, în starea ST_3 se face o citire din memorie şi datele citite sunt încărcate în registrul M (*fetch* operand).

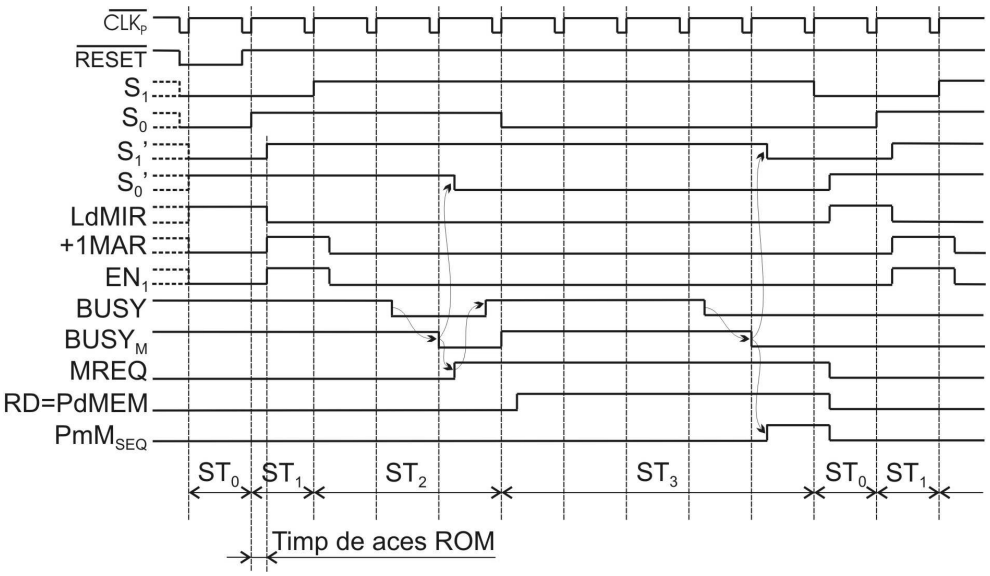


Fig 3.11 Procesarea unei microinstrucţiuni de transfer asincron cu microcomanda READ codificată în câmpul MEMORY OPERATION – cronograma de semnale

Cronograma evidenţiază formele de undă aferente semnalelor de interes (semnale ce reprezintă starea prezentă, starea următoare, intrări şi respectiv ieşiri din automat). În cronogramă este redat ciclul microinstrucţiunii în integralitatea lui (toate cele patru stări parcurse de către automatul SEQ). Cronograma porneşte din starea iniţială instalată prin comanda de iniţializare a automatului ($\overline{RESET}$). Întreaga cronogramă a fost construită printr-o simulare funcţională a automatului. Pentru această simulare (pur teoretică) s-au utilizat: organigrama SEQ (figura3.7), schema SEQ (figura3.10) şi conţinutul ROM-ului din componenţa SEQ (tabelul 3.13). Să notăm că astfel de cronograme care simulează funcţionalitatea unui bloc *hardware* se pot obţine automat atunci când proiectarea este realizată într-un mediu de proiectare cum ar fi, de exemplu, VHDL.

Ieşirea din starea ST_3 se face numai când operaţia de citire din memorie se încheie. Încheierea operaţiei de citire este semnalată prin dezectivarea semnalului BUSY. Din starea ST_3, SEQ revine în starea ST_0 (pentru *fetch*-ul următoarei microinstrucţiuni).

Conform organigramei SEQ (figura 3.7), comenzile MREQ, RD şi PdMEM sunt active pe toată durata stării ST_3 (figura 3.11). Spre deosebire de acestea, comanda PmM_{SEQ} se va activa numai pe ultima perioadă de tact din cadrul stării ST_3 (după dezactivarea semnalului BUSY). Dezactivarea semnalului BUSY certifică faptul că datele citite din memorie sunt prezente pe MBUS (figura 2.2); PdMEM, încă activă, face ca datele prezente pe MBUS să fie transmise şi pe RBUS. De aici, cu comanda PmM_{SEQ}, vor fi încărcate în registrul M.

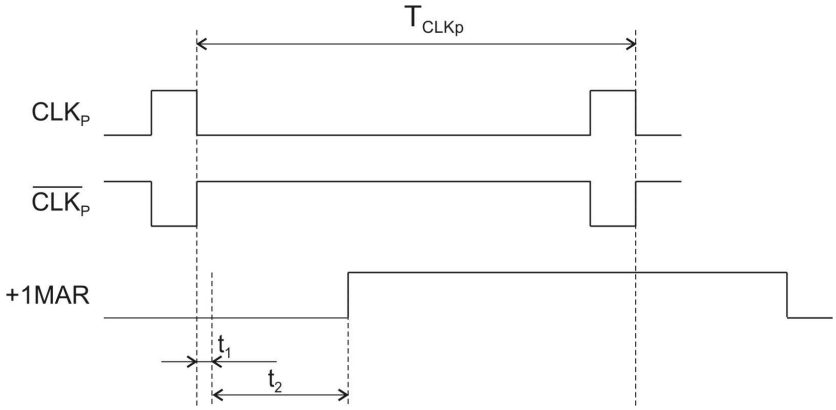
O microinstrucţiune de transfer asincron care conţine microcomanda IFCH codificată în câmpul MEMORY OPERATION este procesată întro manieră similară. Singura deosebire care apare se referă la faptul că, în locul comenzii PmM_{SEQ} se activează comanda $PmIR_{SEQ}$ (microcomanda IFCH am dedicat-o operaţiei de *fetch* instrucţiune care, în ultimă instanţă, reprezintă tot o citire din memorie).

Dacă microinstrucţiunea de transfer asincron conţine microcomanda WRITE codificată în câmpul MEMORY OPERATION, atunci, în starea ST_3, SEQ va controla o operaţie de scriere în memorie. Cronograma de semnale pentru o astfel de microinstrucţiune este similară cu cea din figura 3.11, cu următoarele ajustări:

- în locul comenzilor RD şi PdMEM se activează comenzile WRITE şi EN_3 care au aceeaşi formă de undă (active pe totă durata stării ST_3).

- comanda PmM_{SEQ} nu se activează (vezi organigrama din figura 3.7 şi conţinutul ROM-ului din tabelul 3.13)

ROM-ul din structura automatului SEQ (figura 3.10) introduce o întârziere suplimentară în generarea comenzilor SEQ (întârziere egală cu timpul de acces al ROM-ului). Această întârziere suplimentară se adaugă la întârzierea indusă de timpul de încărcare paralelă în registrul R (timpul scurs de la frontul activ al semnalului de tact $\overline{CLK_P}$ până la fronturile (tranziţiile) provocate, de încărcarea paralelă în discuţie, pe ieşirile de date aferente registrului R. Evident că timpul de acces la ROM este semnificativ mai mare decât timpul de încărcare paralelă în registrul R (figura 3.12).



t_1 –timp de încărcare paralelă în registrul R
t_2 –timpul de acces al memoriei ROM
$t=t_1+t_2$ –întârzierea introdusă în generarea comenzilor

Fig 3.12 Generarea întârziată a comenzilor: exemplificare pe comanda +1MAR

Comenzile generate de către automat vor fi mai „**ferm**" sincronizate cu semnalul de tact al automatului dacă poziţiile celor două blocuri (R şi respectiv ROM) din structura automatului vor fi inversate. În figura 3.13 este redată schema automatului SEQ rezultată în urma inversării celor două elemente componente din structura sa. Inversarea celor două blocuri din structura automatului nu atrage după sine şi modificarea conţinutului memoriei ROM. Prin urmare, tabelul 3.13 (care rezuma conţinutul memoriei ROM din componenţa automatului din figura 3.10) este valabil şi pentru automatul din figura 3.13.

Într-o manieră similară cu cea prezentată în cronograma din figura 3.11, cronograma din figura 3.14 simulează funcţionarea automatului SEQ din figura 3.13, atunci când acesta controlează procesarea unei microinstrucţiuni de transfer asincron cu microcomanda READ codificată în câmpul MEMORY OPERATION. Am optat pentru simularea funcţională a noului automat pe acelaşi tip de microinstrucţiune pentru ca cititorul să poată evalua mai uşor analogiile şi respectiv diferenţele care apar între cele două variante de implementare.

Să observăm că pentru comenzile obţinute pe ieşirile de date aferente ROM-ului din figura 3.13, am utilizat indicele "**p**" deoarece acestea nu sunt ieşirile reale ale automatului. Pe următorul front urcător al semnalului de tact $\overline{CLK_P}$, aceste comenzi vor fi memorate în registrul R devenind astfel comenzi reale (ieşiri ale automatului). Prin urmare:

- $LdMIR_p$, prin memorarea în registru, devine comanda reală LdMIR
- $LdMAR_p$, prin memorarea în registru, devine comanda reală LdMAR
- $+1MAR_p$, prin memorarea în registru, devine comanda reală +1MAR

  .....
  .....

- $PmMSEQ_p$, prin memorarea în registru, devine comanda reală PmM_{SEQ}

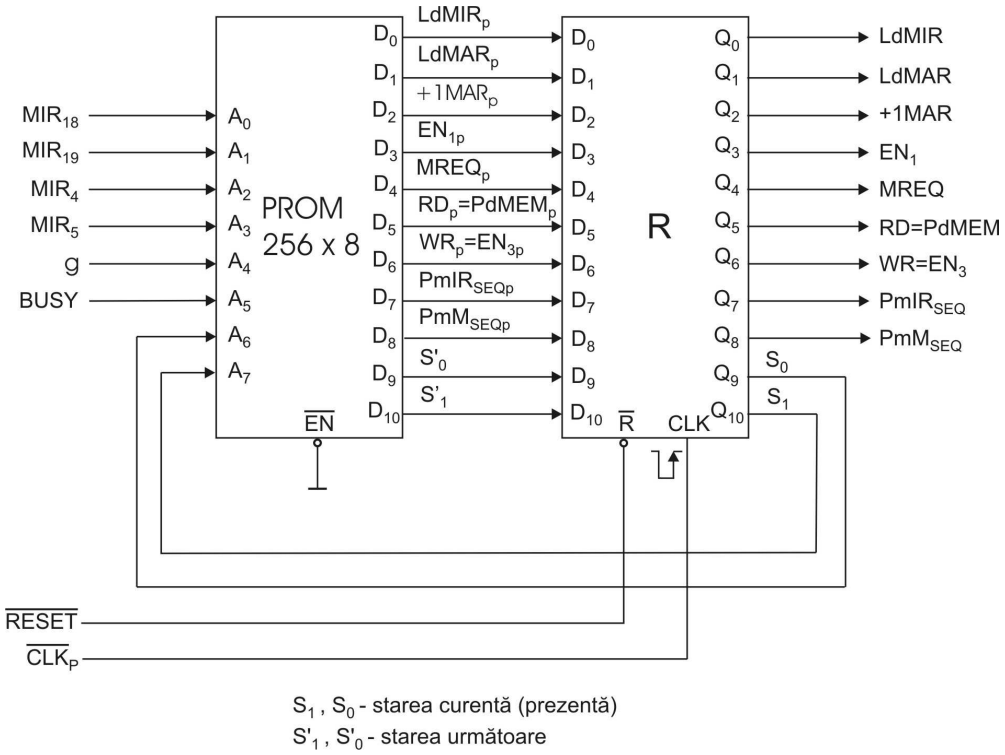


Fig 3.13 Automatul SEQ implementat cu ROM urmat de registru

Aceste perechi de semnale sunt evidenţiate şi în cronograma de semnale din figura 3.14.

În figura 3.15 sunt evidenţiate mai clar întârzierile care apar, luând ca exemplu perechea de comenzi $+1MAR_P$ (intermediară) şi respectiv +1MAR (finală). Se observă din această figură că, timpul de acces al memoriei ROM (t_2) intervine ca întârziere la generarea comenzii intermediare $+1MAR_P$ iar timpul de încărcare paralelă în registrului R (t_1) intervine ca întârziere la generarea comenzii finale +1MAR. Deşi se activează cu o perioadă de tact întârziere, comanda finală (+1MAR) este „**mult mai sincronă**” cu frontul activ al semnalului de tact aferent automatului ($\overline{CLK_P}$). Întârzierea de o perioadă de tact între comanda întermediară şi comanda finală (rezultată din cea intermediară), este valabilă pentru toate comenzile elaborate de automat. Să menţionăm că această întârziere nu deranjează din punct de vedere funcţional deoarece, în mediul extern (mediul controlat de automat) intervin doar comenzile finale. Sincronizarea „**mult mai fermă**” a comenzii finale cu tactul automatului poate fi mai importantă şi în anumite aplicaţii poate reprezenta un câştig esenţial.

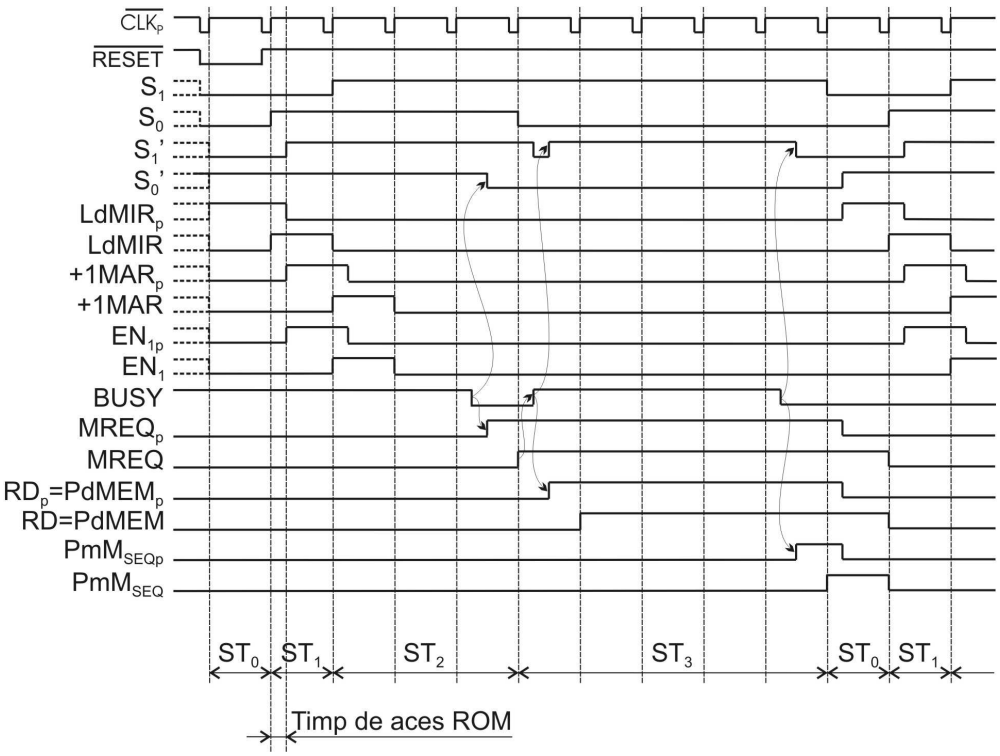


Fig 3.14 Procesarea unei microinstrucţiuni de transfer asincron cu microcomanda READ codificată în câmpul MEMORY OPERATION – cronograma de semnale aferentă automatului din figura 3.13

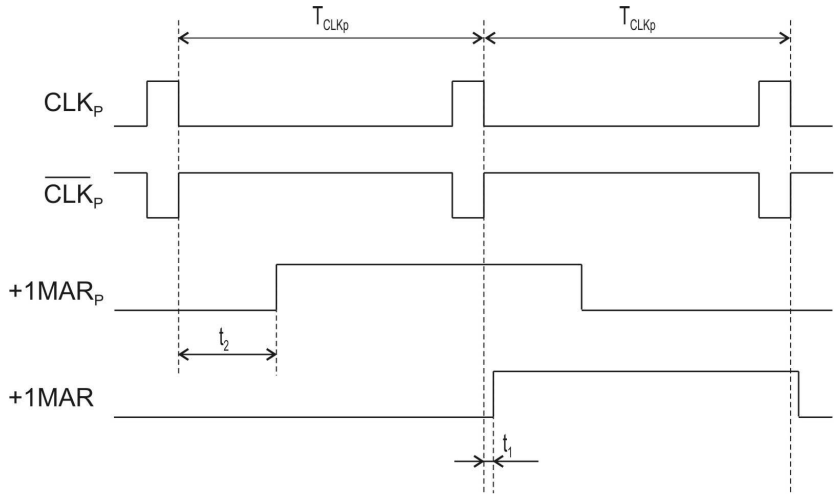


Fig 3.15 Întârzierile introduse în generarea comenzilor de către automatul din figura 3.13: exemplificare pe comanda +1MAR

Comenzile finale generate de către automatul SEQ (atât cel din figura 3.10 cât şi cel din figura 3.13) sunt comenzi de tip nivel (active pe durata unei întregi perioade de tact). Acolo unde este cazul, din comanda de tip nivel se poate obţine comanda de tip impuls corespondentă. Obţinerea comenzii de tip impuls este ilustrată în figura 3.16 unde, din comanda de tip nivel PmM_{SEQ} se obţine comanda de tip impuls $PmM_{SEQ\text{-}impuls}$.

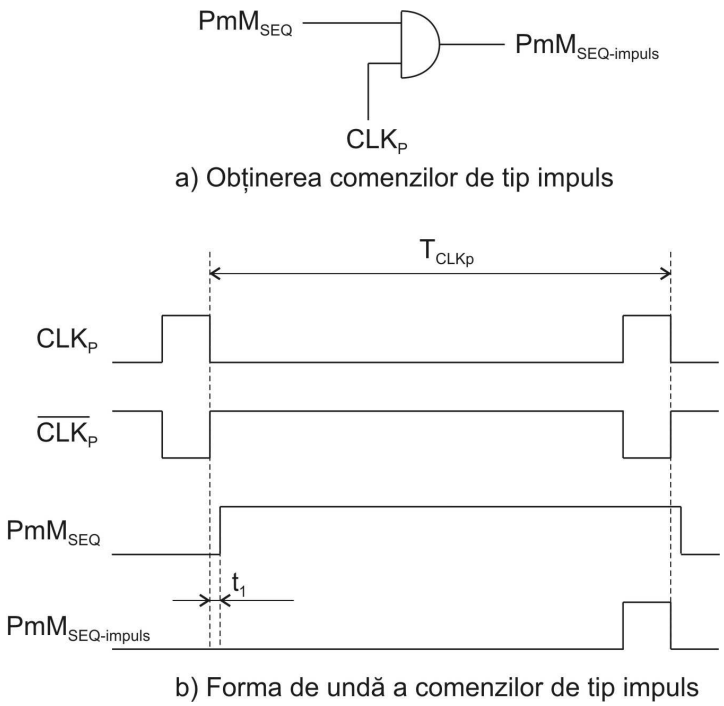


Fig 3.16 Generarea comenzilor de tip impuls

<u>Observaţia 1</u>:

Semnalul de iniţializare ($\overline{RESET}$ *) are rolul de a iniţializa procesorul precum şi celelalte blocuri funcţionale din structura calculatorului; toate blocurile funcţionale care revendică pornirea dintr-o stare iniţială bine definită, vor fi aduse în această stare iniţială, imediat după punerea sub tensiune a calculatorului. Să enumerăm câteva dintre operaţiile realizate de comanda de iniţializare* ($\overline{RESET}$):

a) Iniţializarea automatului SEQ (aducerea automatului în starea ST_0*).*

b) Încărcarea constantei zero în registrul MAR.

c) *Încărcarea unei constante bine dfinite în registrul PC (adresa unei locaţii din zona de ROM a memoriei principale).*

d) *Iniţializarea sistemului de întreruperi şi excepţii (SIE).*

e) *Iniţializarea controller-elor I/O aferente perifericelor.*

Acţiunile a) şi b) obligă automatul SEQ să-şi înceapă activitatea cu fetch-ul microinstrucţiunii aflată în MPM la microadresa zero (fetch-ul primei microinstrucţiuni din cadrul microrutinei Fetch Instrucţiune).

Acţiune c) obligă procesorul să-şi înceapă activitatea cu fetch-ul unei instrucţiuni din ROM (la IBM-PC va fi prima instrucţiune din BIOS). Funcţionarea calculatorului, imediat după punerea sub tensiune, nu ar putea starta cu fetch-ul unei instrucţiuni din RAM deoarece în RAM nu există încă un program. Celulele de memorare din zona RAM, la punerea sub tensiune, se poziţionează aleator pe unu sau pe zero (hazardul provocat de punerea sub tensiune a calculatorului). De abia după încărcarea unui prim program în RAM (de pe disc sau din ROM), se poate starta procesarea instrucţiunilor din RAM.

Acţunile d) şi e) aduc în stare iniţială sistemul de întreruperi şi excepţii şi respectiv controller-ele I/O aferente perifericelor.

În cronogramele din figurile 3.11 şi 3.14 am presupus că semnalul $\overline{RESET}$ *, activ pe zero, este sincronizat cu tactul procesorului (*CLK_P*). Pe de altă parte, semnalul (comanda sincronă)* $\overline{RESET}$ *se va obţine din comanda asincronă* $\overline{RESET}_A$ *(figura 3.20) care, la rândul ei, se generează din două surse (figura 3.17):*

a) *La punerea sub tensiune a calculatorului (iniţializarea la pornire cu ajutorul semnalului* $\overline{RESET}_P$ *)*

b) *La acţionarea butonului RESET de către utilizator (iniţializarea din buton realizată de semnalul* $\overline{RESET}_B$ *)*

Comanda globală $\overline{RESET}_A$ *(reset asincron) se obţine prin reunirea celor două surse de generare. Într-adevăr, poarta SI din figura 3.17 realizează funcţia:*

$$\overline{RESET}_A = \overline{RESET}_P \bullet \overline{RESET}_B$$

Dacă se aplică teorema 2 a lui De Morgan ($\overline{A+B} = \overline{A} \bullet \overline{B}$*) termenului drept şi apoi se neagă ecuaţia, se obţine:*

$$RESET_A = RESET_P + RESET_B$$

Termenul drept al acestei ultime ecuaţii conţine o funcţie SAU care realizează reunirea celor două surse de generare a comenzii asincrone $RESET_A$*. Vom analiza în continure generarea semnalelor* $\overline{RESET}_P$ *şi respectiv* $\overline{RESET}_B$ *(figura 3.17).*

La punerea sub tensiune a calculatorului, sursa va genera tensiunile continue de alimentare. Tensiunea de +5V, de exemplu, va alimenta un circuit R-C demarând astfel

procesul de încărcare a condensatorului C cu o sarcină electrică Q_C (figura 3.17). Tensiunea de încărcare U_C variază în timp ca în figura 3.18 (valorile parametrilor R şi respectiv C vor determina timpul de încărcare). Atâta timp cât $U_C<U_{pragTS}$ (U_{pragTS} reprezintă tensiunea de prag a porţii Trigger-Smidt), atunci ieşirea acestei porţi (notată cu A în figura 3.17) va fi pe "0" (zero logic). În momentul în care se atinge egalitatea ($U_C=U_{pragTS}$), ieşirea porţii A urcă în "1" (unu logic) şi va rămâne în "1" pe toată durata funcţionării calculatorului. În această manieră se obţine semnalul $\overline{RESET}_P$ a cărui formă de undă este redată în figura 3.18.

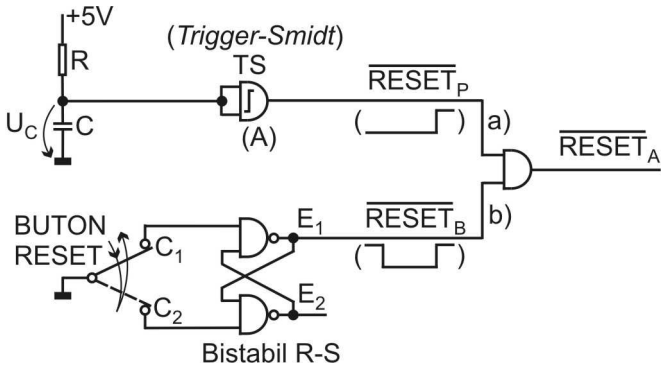


Fig 3.17 Generarea semnalului $\overline{RESET}_A$
a) La punerea sub tensiune a calculatorului
b) La acţionarea butonului RESET

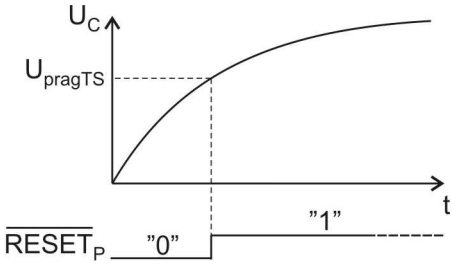
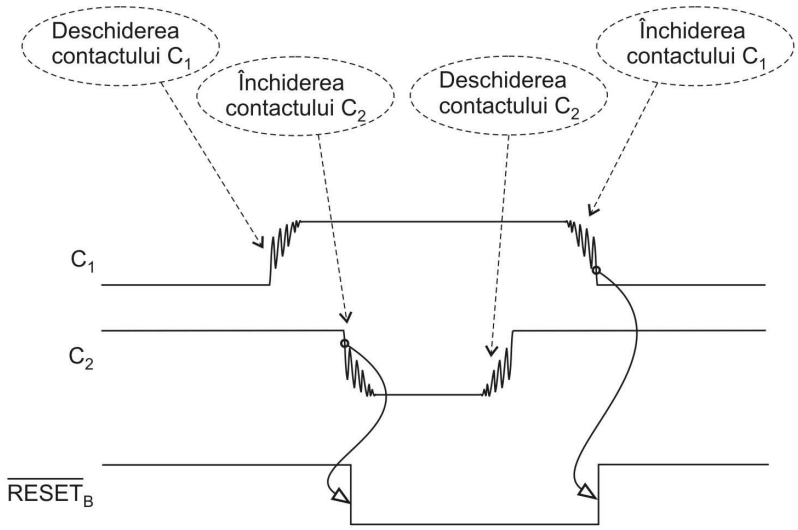


Fig 3.18 Generarea semnalului $\overline{RESET}_P$ la punerea sub tensiune a calculatorului.

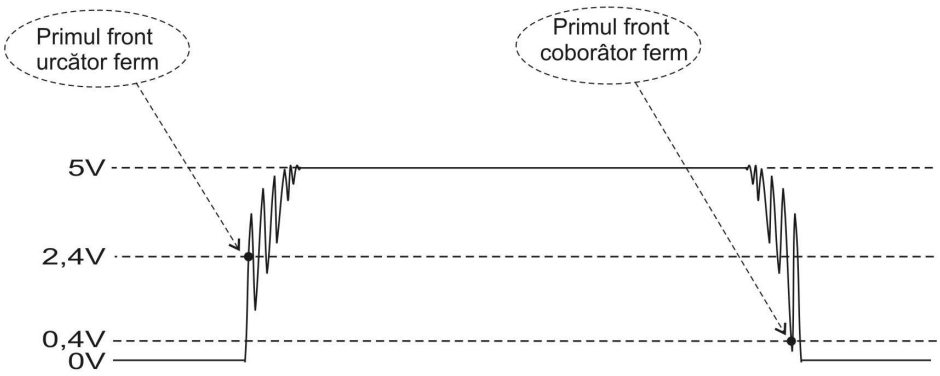
Pe de altă parte, acţionarea butonului RESET de către operator (figura 3.17) va produce comutarea, urmată de recomutarea bistabilului R-S implementat cu cele două porţi SI-NU. În stare normală (de repaus), butonul RESET menţine închis contactul C_1 şi respectiv, deschis contactul C_2. Prin urmare, în stare de repaus ieşirea E_1 va fi în "1". La apăsarea butonului RESET, mai întâi se deschide contactul C_1 şi apoi se închide contactul C_2. La

închiderea contactului C_2, bistabilul comută în starea complementară (ieşirea E_1 trece în "0"). La eliberarea butonului, mai întâi se deschide contactul C_2 şi apoi se închide C_1, butonul RESET ajungând astfel din nou în starea de repaus. La închiderea contactului C_1, bistabilul revine în starea iniţială (ieşirea E_1 revine în "1"). Prin urmare, durata impulsului în "0" obţinut pe ieşirea E_1 (semnalul $\overline{RESET}_B$) este egală cu durata de timp în care butonul RESET este menţinut apăsat.

Trebuie subliniat faptul că bistabilul R-S realizează şi o filtrare a oscilaţiilor electrice produse de vibraţiile mecanice ale contactelor C_1 şi C_2 (figura 3.19.a).



a) oscilațiile electrice produse de vibrațiile mecanice ale contactelor C_1 si C_2



b) fronturi ferme pentru comutare în "1" şi respectiv în "0" (exemplificare)

Fig. 3.19 Oscilaţiile electrice produse de vibraţiile contactelor mecanice

Semnalele electrice obţinute pe contactele C_1 şi C_2 (pe intrările bistabilului R-S), nu vor avea fronturi singulare şi ferme; la tranziţia de la un nivel logic la alt nivel logic, aceste semnale vor prezenta oscilaţii electrice datorate vibraţiilor mecanice produse la acţionarea celor două contacte (figura 3.19.a).).

Orice contact mecanic, la acţionare vibrează şi produce astfel de oscilaţii electrice. Bistabilul R-S din figura 3.18 comută atunci când pe intrările C_2 şi respectiv C_1 apar tranziţii din "1" în "0" (figura 3.19.a).). Semnalul $\overline{RESET}_B$ obţinut pe ieşirea E_1 aferentă bistabilului R-S va avea însă fronturi ferme deoarece frontul primei oscilaţii ferme de pe intrarea C_1 (C_2) va provoca comutarea bistabilului (comutarea ieşirii E_1) iar următoarele oscilaţii de pe inrarea C_1 (C_2) nu mai produc niciun efect asupra ieşirii E_1. Prin oscilaţie fermă trebuie să înţelegem acea oscilaţie care generează un impuls capabil de comutare (un impuls care are calitatea de a atinge pragul de comutare). Pentru tehnologia TTL, unde pragurile de comutare sunt 0,4V (pentru comutarea în "0") şi respectiv 2,4V (pentru comutarea în "1"), fronturile ferme sunt explicitate în figura 3.19.b). Pentru tehnologia MOS fenomenele sunt identice; diferă doar pragurile de comutare. Bistabilul R-S din figura 3.18 reacţionează (comută) doar la tranziţiile high-low care apar pe intrările C_2 şi respectiv C_1; tranziţiile low-high de pe intrările C_2 şi respectiv C_1 nu determină comutarea bistabilului (figura 3.19.b).).

În concluzie, atât semnalul $\overline{RESET}_B$ cât şi semnalul $\overline{RESET}_P$ vor avea fronturi high-low şi respectiv low-high singulare şi ferme dar aceste fronturi vor fi asincrone (nesincronizate cu semnalul de tact al procesorului CLK_P). Prin urmare şi semnalul global de iniţializare $\overline{RESET}_A$ (reset asincron), care se obţine prin reunirea celor două semnale de iniţializare ($\overline{RESET}_B$ şi respectiv $\overline{RESET}_P$), va fi tot asincron în raport cu tactul procesorului. Frontul de dezactivare a comenzii globale $\overline{RESET}_A$ (frontul urcător) trebuie sincronizat şi transformat într-o comandă de start sincronizată cu tactul CLK_P. Această sincronizare se poate realiza cu ajutorul a doi bistabili conectaţi în cascadă, ca în figura 3.20.

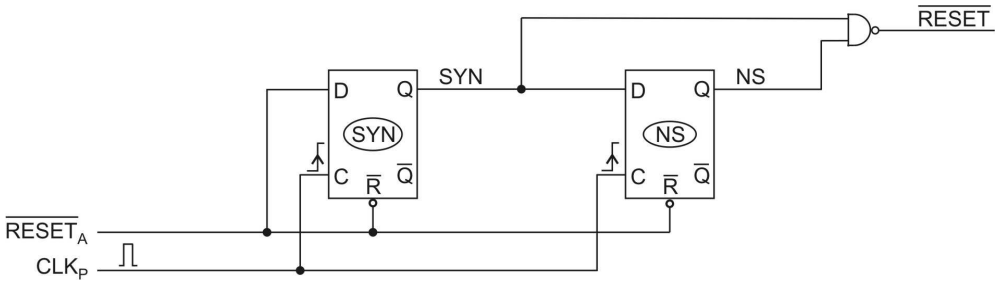


Fig 3.20 Sincronizarea comenzii asincrone $\overline{RESET}_A$ cu semnalul de tact al procesorului CLK_P

Notaţiile introduse în figura 3.20 sunt:

SYN –bstabil de sincronizare

NS –bstabil de generare nivel sincron (generează un semnal de tip nivel cu durata egală cu o perioadă de tact $TCLK_P$).

Cronograma de semnal din dinfigura 3.21 ilustrează funcţionarea schemei din figura 3.20.

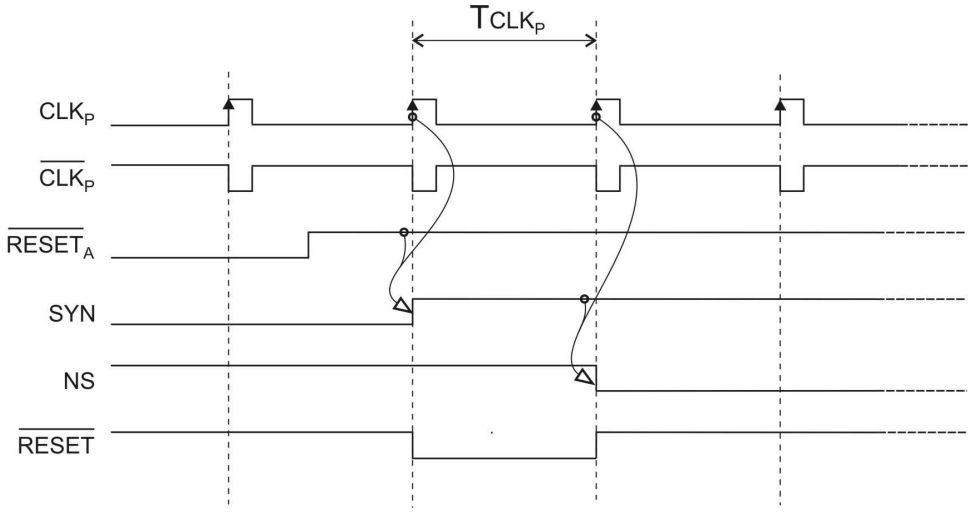


Fig 3.21 Generarea semnalului $\overline{RESET}$ sincronizat cu tactul CLK_P

Primul front urcător de pe semnalul CLK_P, care survine după dezactivarea comenzii asincrone $\overline{RESET}_A$ (după eliberarea celor doi bistabili din reset), va seta sincron bistabilul SYN iar următorul front, va seta sincron bistabilul NS. Conform figurii 3.20, semnalul $\overline{RESET}$ se obţine printr-o funcţie SI-NU:

$$\overline{RESET} = \overline{SYN \bullet NS}$$

Din figura 3.21 se observă că semnalul $\overline{RESET}$ este sincronizat cu tactul procesorului (CLK_P) şi are chiar forma considerată în cronogramele 3.11 şi 3.14. Prin urmare, sincronizarea comenzii asincrone $\overline{RESET}_A$ a fost realizată.

***Observaţia 2**:*
Cronograma din figura 3.11 ilustreză funcţionarea automatului SEQ din figura 3.10. Analizând forma de undă a semnalului LdMIR constatăm că acesta se activează periodic pe durata stării ST_0 a automatului SEQ; în fiecare stare ST_0 se obţine un front urcător pe

semnalul LdMIR, front care va încărca, în registrul MIR, microinstrucţiunea extrasă din locaţia MPM selectată de microadresa prezentă în registrul MAR (fetch-ul microinstrucţiunii).

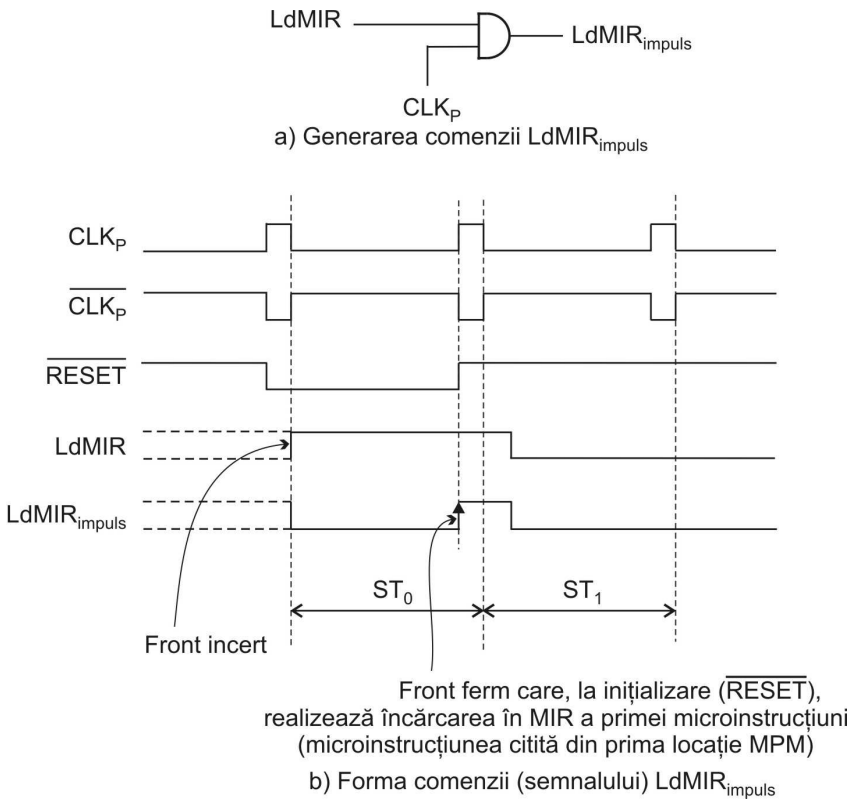


Fig 3.22 Schema de generare a comenzii LdMIR$_{impuls}$
şi forma de undă a acestei comenzi

Dacă analizăm mai atent comportamentul la iniţializare al automatului SEQ din figura 3.10 (comportamentul la activarea comenzii $\overline{RESET}$ *), constatăm că primul front urcător de pe semnalul LdMIR (vezi cronograma din figura 3.11) nu este sigur; într-adevăr, dacă bistabilii Q_7 şi Q_6 din registrul R (bistabilii care memorează starea prezentă (S_1,S_0) a automatului) se poziţionează pe (0,0) (se resetează) în momentul punerii sub tensiune a procesorului, atunci comanda* $\overline{RESET}$ *(care resetează aceşti bistabili si de altfel, întreg registrul R) nu va mai produce primul front urcător pe semnalul LdMIR (deficienţă). Altfel spus, dacă hazardul care intervine la punerea sub tensiune conduce la Q_7,Q_6=0,0 atunci comanda* $\overline{RESET}$ *, activată ulterior, nu va mai avea niciun efect asupra bistabililor Q_7 şi Q_6 deoarece aceştia au fost resetaţi încă din momentul punerii sub tensiune a procesorului; din acest motiv nu se va mai obţine acel prim front urcător pe*

semnalul LdMIR. Asta înseamnă că încărcarea în MIR a primei microinstrucţiuni (fetch-ul primei microinstrucţiuni) nu se va realiza.

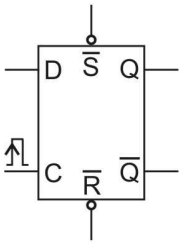
Există evident mai multe soluţii la această problemă. Una dintre acestea constă în utilizarea unei comenzi LdMIR de tip impuls, generată aşa cum se propune în figura 3.22. Impulsurile obţinute pe semnalul $LdMIR_{impuls}$ vor realiza încărcarea în registrul MIR a microinstrucţiunilor extrase din MPM.

Primul front urcător de pe semnalul $LdMIR_{impuls}$ devine un front cert chiar dacă bistabilii Q_7 şi Q_6 se resetează (anticipat) încă din momentul punerii sub tensiune a procesorului (resetare rezultată în urma hazardului care intervine la punerea sub tensiune).

3.5.6 Implementarea automatului SEQ cu bistabil asociat la fiecare stare

Impementarea automatului SEQ cu ROM şi registru (vezi figurile 3.10 şi 3.13) reprezintă o soluţie relativ costisitoare. Planul de memorie ROM utilizat are o dimensiune destul de mare (256 locaţii × 11 biţi). Prin urmare, cantitatea de *hardware* necesară acestei implementări este destul de consistentă. În acest paragraf vom prezenta metodologia de proiectare a automatului SEQ cu bistabil asociat la fiecare stare. Automatele de acest tip se mai numesc şi automate cu stări decodificate sau automate cu stări necodificate.

Automatul SEQ are patru stări (fig 3.7). În cazul implementării cu stări necodificate, fiecărei stări i se asociază un bistabil. Pot fi utilizaţi bistabili D sau JK. Pentru automatul SEQ cu stări necodificate ne propunem să utilizăm bistabili D (figura 3.23). Cum automatul SEQ are patru stări, vor fi necesari patru bistabili.



a) Bistabilul D

D^n	Q^n	Q^{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

b) Tabela de adevăr a bistabilului D

Fig 3.23 Bistabilul D şi tabela sa de adevăr

Reamintim semnificaţia intrărilor şi respectiv ieşirilor aferente bistabilului D:

D -inratrea de date

C -intrarea de tact (clock)

$\overline{S}$ -intrarea de setare asincronă

$\overline{R}$ -intrarea de resetare asincronă

Q -ieşire de date (ieşirea adevărată)

$\overline{Q}$ -ieşire de date (ieşirea negată)

Pe baza tabelei de adevăr din figura 3.23.b), se obţine ecuaţia de stare a bistabilului D:

$$Q^{n+1} = D^n \qquad (3.8)$$

Semnificaţia practică a acestei ecuaţii de stare este: **„pentru ca ieşirea Q să devină "1" în starea următoare (ST_{n+1}) trebuie ca intrarea D să fie pe "1" în starea curentă (ST_n)"**.

Structura de la care porneşte construcţia automatului SEQ cu bistabili D şi stări necodificate este redată în figura 3.24.

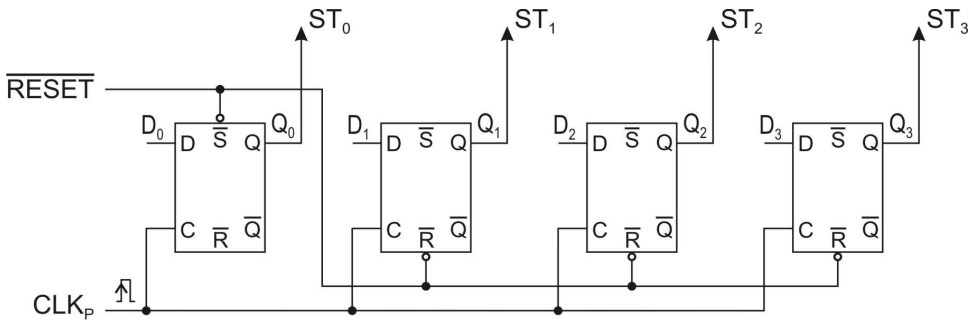


Fig 3.24 Structura de bază de la care porneşte construcţia automatului SEQ cu bistabili D şi stări necodificate

Schema din figura 3.24 relevă faptul că fiecărei stări îi este asociat un bistabil. Automatul va fi în starea ST_i atunci când bistabilul i este setat (atunci când Q_i=1). Asocierea dintre bistabilii din componenţa automatului şi stările acestuia este prezentată în tabelul 3.14.

Q_0	Q_1	Q_2	Q_3	Starea
1	0	0	0	$ST_0=Q_0$
0	1	0	0	$ST_1=Q_1$
0	0	1	0	$ST_2=Q_2$
0	0	0	1	$ST_3=Q_3$

Tabelul 3.14 Asocierea stare-bistabil pentru automatul SEQ cu stări necodificate

A. Graful de tranziţii

Celor patru stări din graful de tranziţii al automatului SEQ (fig 3.8) le asignăm bistabilii asociaţi (tabelul 3.14). Se obţine astfel graful de tranziţii din figura 3.25

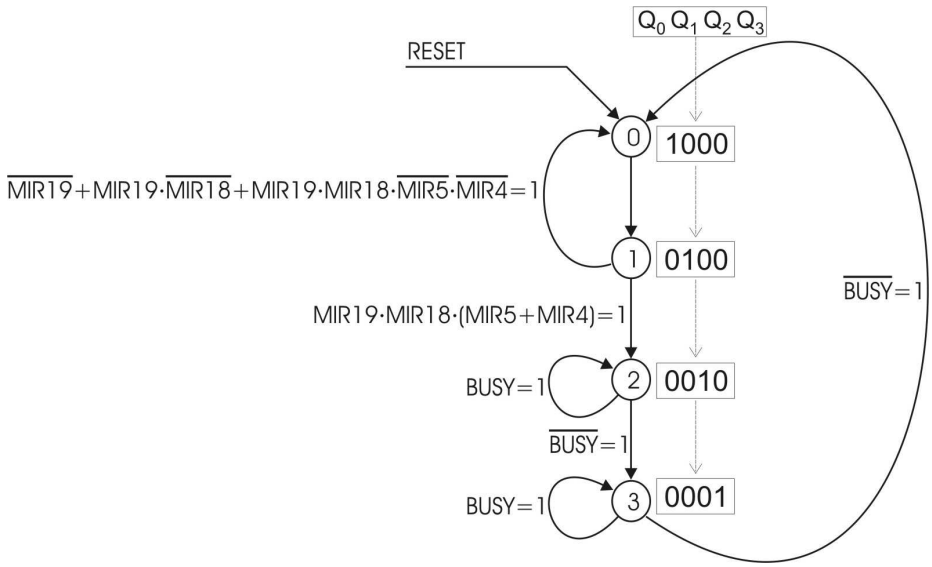


Fig 3.25 Graful de tranziţii aferent automatului SEQ cu stări necodificate

B. Ecuaţiile intrărilor D_0, D_1, D_2 şi D_3

Tabela de adevăr din figura 3.23 conţine patru rânduri şi fiecare rând sintetizează o tranziţie $Q^n \rightarrow Q^{n+1}$. Tranziţia sintetizată pe rândul 3 al tabelei de adevăr generează <u>regula 1 obligatorie</u> pentru intrarea de date D:

<u>Regula 1</u>: Dintr-o stare curentă $Q^n = 0$, bistabilul D comută într-o stare următoare $Q^{n+1} = 1$ numai dacă, pe durata stării curente, intrarea de date este pe ”1” ($D^n = 1$).

Tranziţia sintetizată pe rândul 4 al tabelei de adevăr generează <u>regula 2 obligatorie</u> pentru inrarea de date D:

<u>Regula 2</u>: Dintr-o stare curentă $Q^n = 1$, bistabilul D comută într-o stare următoare $Q^{n+1} = 1$ numai dacă, pe durata stării curente, intrarea de date este pe ”1” ($D^n = 1$).

Cele două reguli sintetizează tranziţiile pentru care semnalul aplicat pe intrarea D trebuie să aibă valoarea ”1”. Celelalte două tranziţii din tabela de adevăr 2.23.b) se realizează cu ”0” (zero logic) aplicat pe intrarea D.

Prin urmare, pentru a scrie ecuaţiile intrărilor D_0, D_1, D_2 şi D_3, în graful de tranziţii din figura 3.25, trebuie căutate tranziţiile $Q^n = 0 \rightarrow Q^{n+1} = 1$ şi respectiv $Q^n = 1 \rightarrow Q^{n+1} = 1$; pentru a realiza aceste tranziţii trebuie să se asigure $D^n = 1$.

a) Ecuaţia intrării D_0

Bistabilul 0 (ieşirea Q_0) are următoarele tranziţii din ”0” în ”1” (figura 3.25):

- La tranziţia automatului SEQ din starea ST_1 în starea ST_0 (tranziţie condiţionată de condiţia: $\overline{MIR_{19}} + MIR_{19} \bullet \overline{MIR_{18}} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR_5} \bullet \overline{MIR_4} = 1$).

  Această tranziţie induce în ecuaţia intrării D_0, termenul (produsul):

  $$ST_1 \bullet (\overline{MIR_{19}} + MIR_{19} \bullet \overline{MIR_{18}} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR_5} \bullet \overline{MIR_4}) =$$
  $$= Q_1 \bullet (\overline{MIR_{19}} + MIR_{19} \bullet \overline{MIR_{18}} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR_5} \bullet \overline{MIR_4})$$

- La tranziţia automatului din starea ST_3 în starea ST_0 (tranziţie condiţionată de condiţia $\overline{BUSY} = 1$). Această tranziţie induce în ecuaţia intrării D_0, termenul (produsul):

  $$ST_3 \bullet \overline{BUSY} = Q_3 \bullet \overline{BUSY}$$

Analizând toate tranziţiile din graful de tranziţii prezentat în figura 3.25, constatăm că bistabilul 0 (ieşirea Q_0) nu are tranziţii din ”1” în ”1”. Prin urmare, ecuaţia intrării D_0 va reuni doar cele două produse induse de tranziţiile din ”0” în ”1”:

$$D_0 = Q_1 \bullet (\overline{MIR_{19}} + MIR_{19} \bullet \overline{MIR_{18}} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR_5} \bullet \overline{MIR_4}) + Q_3 \bullet \overline{BUSY} \quad (3.9)$$

b) Ecuaţia intrării D_1

Bistabilul 1 (ieşirea Q_1) are o singură tranziţie din ”0” în ”1” (figura 3.25); aceasta are loc la tranziţia automatului SEQ din starea ST_0 în starea ST_1 (tranziţie necondiţionată). În ecuaţia intrării D_1, această tranziţie induce termenul: ST_0=Q_0.

Din acelaşi graf de tranziţii (figura 3.25) constatăm că bistabilul 1 (ieşirea Q_1) nu are tranziţii din ”1” în ”1”. Prin urmare, ecuaţia intrării D_1 va fi:

$$D_1 = Q_0 \quad (3.10)$$

c) Ecuaţia intrării D_2

Bistabilul 2 (ieşirea Q_2) are o singură tranziţie din ”0” în ”1” (figura 3.25); aceasta are loc la tranziţia automatului SEQ din starea ST_1 în starea ST_2 (tranziţie condiţionată de condiţia $MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) = 1$). În ecuaţia intrării D_2, această tranziţie induce termenul (produsul):

$$ST_1 \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) = Q_1 \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) =$$
$$= Q_1 \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_5 + Q_1 \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_4$$

De asemenea, bistabilul 2 (ieşirea Q_2) are tot o singură tranziţie din ”1” în ”1” (figura 3.25); aceasta are loc la tranziţia automatului SEQ din starea ST_2 în aceeaşi stare ST_2 (tranziţie condiţionată de condiţia BUSY=1). În ecuaţia intrării D_2, această tranziţie induce termenul (produsul):

$$ST_2 \bullet BUSY = Q_2 \bullet BUSY$$

Prin urmare, ecuaţia intrării D_2 se obţine reunind (printr-o funcţie SAU) produsele generate de aceste ultime două tranziţii ale automatului SEQ:

$$D_2 = Q_1 \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_5 + Q_1 \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_4 + Q2 \bullet BUSY \qquad (3.11)$$

d) Ecuaţia intrării D_3

Pe baza grafului de tranziţii din figura 3.25 constatăm că bistabilul 3 (ieşirea Q_3) are o singură tranziţie din ”0” în ”1” (la tranziţia automatului SEQ din starea ST_2 în starea ST_3; tranziţie condiţionată de condiţia $\overline{BUSY} = 1$) şi o singură tranziţie din ”1” în ”1” (la tranziţia automatului SEQ din starea ST_3 în aceeaşi stare ST_3; tranziţie condiţionată de condiţia BUSY=1). Prin urmare, ecuaţia intrării D_3 se va obţine reunind (prin funcţie SAU) termenii (produsele) generate de cele două tranziţii:

$$D_3 = ST_2 \bullet \overline{BUSY} + ST_3 \bullet BUSY = Q_2 \bullet \overline{BUSY} + Q_3 \bullet BUSY \qquad (3.12)$$

În concluzie, ecuaţiile intrărilor sunt:

$$\begin{aligned} D_0 &= Q_1 \bullet (\overline{MIR_{19}} + MIR_{19} \bullet \overline{MIR_{18}} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR_5} \bullet \overline{MIR_4}) + Q_3 \bullet \overline{BUSY} \\ D_1 &= Q_0 \\ D_2 &= Q_1 \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_5 + Q_1 \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_4 + Q2 \bullet BUSY \\ D_3 &= Q_2 \bullet \overline{BUSY} + Q_3 \bullet BUSY \end{aligned} \qquad (3.13)$$

C. Ecuaţiile ieşirilor

Ieşirile unui automat reprezintă comenzile pe care acesta le generează. Prin comenzile pe care le generează (vezi figura 3.7), automatul SEQ controlează operaţiile interne unităţii de control microprogramate (*fetch*-ul şi execuţia microinstrucţiunilor, în stările ST_0 şi respectiv ST_1) precum şi operaţiile de transfer procesor-memorie (citirea din şi respectiv scrierea în memorie, în stările ST_2 şi ST_3). Pe baza organigramei SEQ din figura 3.7, se pot deduce ecuaţiile comenzilor:

$$
\begin{aligned}
&LdMIR = ST_0 \\
&+1MAR = ST_1 \bullet \overline{MIR_{19}} \bullet \overline{g} + ST_1 \bullet MIR_{19} \\
&LdMAR = ST_1 \bullet \overline{MIR_{19}} \bullet g \\
&EN_1 = ST_1 \bullet MIR_{19} \\
&MREQ = ST_2 \bullet \overline{BUSY} + ST_3 \\
&WR = EN_3 = ST_3 \bullet MIR_5 \bullet MIR_4 \\
&RD = PdMEM = ST_3 \bullet \overline{WRITE} = ST_3 \bullet \overline{MIR_5 \bullet MIR_4} \\
&PmM_{SEQ} = ST_3 \bullet READ \bullet \overline{BUSY} = ST_3 \bullet MIR_5 \bullet \overline{MIR_4} \bullet \overline{BUSY} \\
&PmIR_{SEQ} = ST_3 \bullet IFCH \bullet \overline{BUSY} = ST_3 \bullet \overline{MIR_5} \bullet MIR_4 \bullet \overline{BUSY}
\end{aligned}
\tag{3.14}
$$

Ţinând cont de faptul că fiecărei stări îi este asociat un bistabil (figura 3.24), ecuaţiile (3.14) devin:

$$
\begin{aligned}
&LdMIR = Q_0 \\
&+1MAR = Q_1 \bullet \overline{MIR_{19}} \bullet \overline{g} + Q_1 \bullet MIR_{19} \\
&LdMAR = Q_1 \bullet \overline{MIR_{19}} \bullet g \\
&EN_1 = Q_1 \bullet MIR_{19} \\
&MREQ = Q_2 \bullet \overline{BUSY} + Q_3 \\
&WR = EN_3 = Q_3 \bullet MIR_5 \bullet MIR_4 \\
&RD = PdMEM = Q_3 \bullet \overline{MIR_5 \bullet MIR_4} \\
&PmM_{SEQ} = Q_3 \bullet MIR_5 \bullet \overline{MIR_4} \bullet \overline{BUSY} \\
&PmIR_{SEQ} = Q_3 \bullet \overline{MIR_5} \bullet MIR_4 \bullet \overline{BUSY}
\end{aligned}
\tag{3.15}
$$

D. Schema detaliată a automatului

Schema detaliată a automatului SEQ, în varianta de implementare cu bistabili D şi stări necodificate, se obţine implementând în *hardware* ecuaţiile intrărilor (relaţiile (3.13)) şi respectiv ecuaţiile ieşirilor (relaţiile (3.15)) şi adăugând aceste implementări pe structura de bază din figura 3.24. Schema detaliată este redată în figura 3.26.

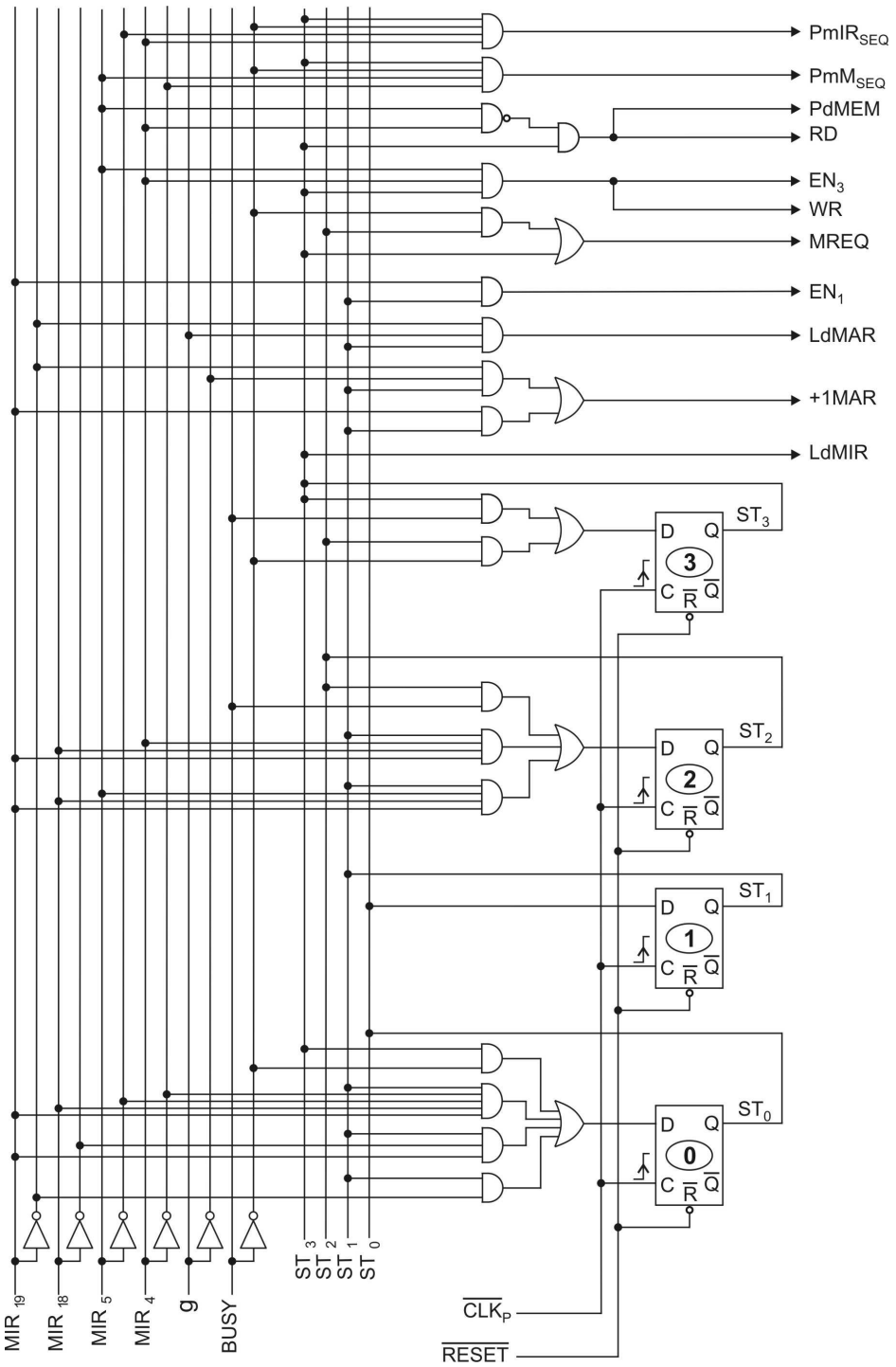


Fig 3.26 Automatul SEQ implementat cu bistabili D şi stări necodificate

3.5.7 Implementarea automatului SEQ cu bistabili şi stări codificate

Pentru codificarea celor patru stări pe care automatul SEQ le tranzitează în timpul funcţionării (fig 3.7), sunt necesari doi bistabili de stare. Vom nota cu **A** şi respectiv **B** cei doi bistabili de stare, cu Q_A şi respectiv Q_B ieşirile celor doi bistabili de stare (figura 3.29) şi vom codifica cele patru stări aşa cum indică tabelul 3.15.

Q_A	Q_B	Stare SEQ
0	0	ST_0
0	1	ST_1
1	1	ST_2
1	0	ST_3

Tabelul 3.15 Codificarea stărilor automatului SEQ

Pentru implementarea automatului SEQ cu stări codificate ne propunem să utilizăm bistabili JK *Master-Slave* (figura 3.27). Reamintim semnificaţia intrărilor şi respectiv ieşirilor bistabilului JK:

J -intrarea de setare (sincronă)

K -intrarea de resetare (sincronă)

C -intrarea de tact (*clock*)

$\overline{S}$ -intrarea de setare (asincronă)

$\overline{R}$ -intrare de setare (asincronă)

Q -ieşirea de date (ieşirea adevărată)

$\overline{Q}$ -ieşirea de date (ieşirea negată)

Dacă bistabilul JK este de tip *master-slave* (JK M-S) atunci eşantionarea intrărilor J şi K se realizează pe frontul urcător al semnalului de tact aplicat pe intrarea C (memorare în bistabilul *master*). Efectul acestei eşantionări, la ieşirile Q şi respectiv $\overline{Q}$, se va vedea doar pe frontul coborâtor al semnalului de tact (transferul stării bistabilului *master* în bistabilul *slave*).

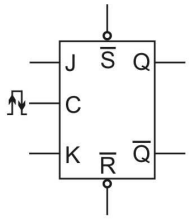
Tabela de adevăr aferentă bistabilului JK M-S este redată în figura 3.27.b). Notaţiile utilizate sunt:

J^n -valoarea aplicată pe intrarea de setare pe durata stării prezente

K^n -valoarea aplicată pe intrarea de resetare pe durata stării prezente

Q^n -starea prezentă (starea bistabilului înaintea impulsului de tact)

Q^{n+1} -starea următoare (starea în care va trece bistabilul în urma acţiunii impulsului de tact, care va eşantiona desigur întrările J şi respectiv K).



a) Bistabilul JK

J^n	K^n	Q^n	Q^{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

b) Tabela de adevăr a bistabilului JK

J^n	K^n	Q^n	Q^{n+1}
0	X	0	0
1	X	0	1
X	1	1	0
X	0	1	1

c) Tabela de adevăr minimizată aferentă bistabilului JK

Fig 3.27 Simbolul şi tabela de adevăr aferentă bistabilului JK

În figura 3.27.c) este redată tabela de adevăr în formă minimizată, care se obţine tin tabela de adevăr extinsă, prin comasarea liniilor care sintetizează comportamente identice ale bistabilului (liniile care prezintă valori identice pe coloanele Q^n şi respectiv Q^{n+1}). Sunt patru perechi de astfel de linii care sintetizează comportamente identice şi acestea sunt marcate cu liniile punctate din figura 3.27.b).

Din tabela de adevăr minimizată rezultă practic direct ecuaţia de stare a bistabilului JK:

$$Q^{n+1} = J \bullet \overline{Q}^n + \overline{K} \bullet Q^n \qquad (3.16)$$

A. Graful de tranziţii

La 3.5.1 a fost prezentată şi explicată organigrama de funcţionare a automatului SEQ (figura 3.7). Pe baza organigramei de funcţionare a rezultat graful de tranziţii al automatului (figura 3.8). Particularizarea acestui graf pentru situaţia în care stările automatului sunt codificate cu ajutorul a doi bistabili de stare (vezi tabelul 3.15) este redată în figura 3.28 iar structura de bază de la care porneşte construcţia automatului SEQ cu bistabili JK şi stări codificate este redată în figura 3.29.

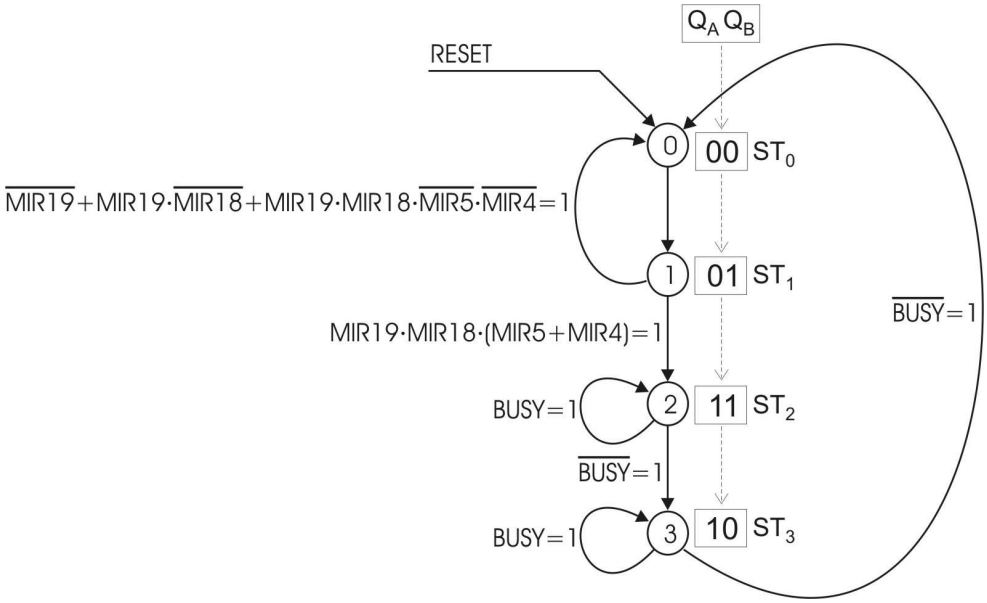


Fig 3.28 Graful de tranziţii al automatului SEQ implementat cu bistabili JK şi stări codificate

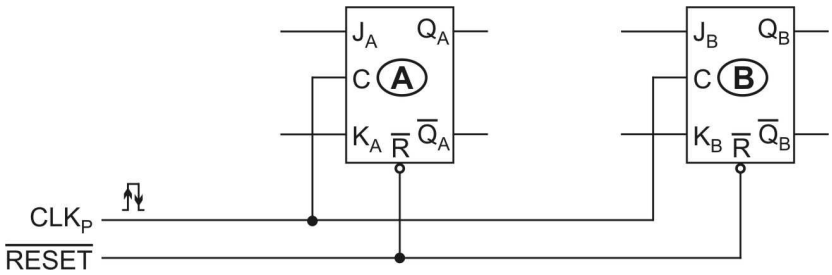


Fig 3.29 Structura de bază de la care porneşte construcţia automatului SEQ implementat cu bistabili JK şi stări codificate

B. Ecuaţiile intrărilor J_A, K_A, J_B şi K_B

Tabela de adevăr din figura 3.27.c) sintetizează funcţionarea bistabilului JK. Această tabelă minimizată conţine patru rânduri şi fiecare rând sintetizează o tranziţie $Q^n \rightarrow Q^{n+1}$.

Tranziţia sintetizată pe rândul 3 din tabelă generează <u>regula 1 obligatorie</u> pentru intrarea de setare (J):

<u>Regula 1</u>: Dintr-o stare curentă $Q^n = 0$, bistabilul JK comută într-o stare următoare $Q^{n+1} = 1$ numai dacă, pe durata stării curente, intrarea de setare este pe ”1” ($J^n = 1$).

Tranziţia sintetizată pe rândul 4 din tabela de adevăr generează <u>regula 2 obligatorie</u> pentru inrarea de resetare (K):

<u>Regula 2</u>: Dintr-o stare curentă $Q^n = 1$, bistabilul JK comută într-o stare următoare $Q^{n+1} = 0$ numai dacă, pe durata stării curente, intrarea de resetare este pe ”1” ($K^n = 1$).

***<u>Observaţia 1</u>*:**
Tranziţiile sintetizate pe rândurile 3 şi respectiv 4 din tabela de adevăr minimizată (figura 3.27.c) nu precizează o valoare obligatorie pentru intrarea J. Prin urmare, cele două tranziţii pot fi realizate atât cu $J^n = 0$ cât şi cu $J^n = 1$. În aceste condiţii, aceste două tranziţii generează <u>două reguli opţionale</u>:

<u>Regula 3</u>: Tranziţia bistabilului JK din starea curentă $Q^n = 1$ în starea următoare $Q^{n+1} = 0$ nu revendică o valoare obligatorie pentru intrarea J^n; prin urmare această tranziţie se realizează corect şi cu $J^n = 1$.

<u>Regula 4</u>: Tranziţia bistabilului JK din starea curentă $Q^n = 1$ în starea următoare $Q^{n+1} = 1$ nu revendică o valoare obligatorie pentru intrarea J^n; prin urmare această tranziţie se realizează corect şi cu $J^n = 1$.

Observaţia 2:
Tranziţiile sintetizate pe rândurile 1 şi respectiv 2 din tabela de adevăr minimizată (figura 3.27.c) nu precizează o valoare obligatorie pentru intrarea K. Prin urmare, cele două tranziţii pot fi realizate atât cu $K^n = 0$ *cât şi cu* $K^n = 1$. *În aceste condiţii, aceste două tranziţii generează încă* *două reguli opţionale*:

Regula 5: Tranziţia bistabilului JK din starea curentă $Q^n = 0$ în starea următoare $Q^{n+1} = 0$ nu revendică o valoare obligatorie pentru intrarea K^n; prin urmare această tranziţie se realizează corect şi cu $K^n = 1$.

Regula 6: Tranziţia bistabilului JK din starea curentă $Q^n = 0$ în starea următoare $Q^{n+1} = 1$ nu revendică o valoare obligatorie pentru intrarea K^n; prin urmare această tranziţie se realizează corect şi cu $K^n = 1$.

În concluzie, pentru elaborarea ecuaţiei intrării J, se va utiliza regula 1 (obligatorie) şi regulile opţionale 3 şi 4. În cadrul ecuaţiei intrării J, regula 1 generează termeni (produse logice) obligatorii iar regulile 3 şi 4 generează termeni opţionali. Aceşti termeni opţionali se pot reţine sau se pot elimina. Scopul este de a reţine doar acei termeni opţionali care, alăturaţi termenilor obligatorii, conduc la minimizarea ecuaţiei intrării J.

Pentru elaborarea ecuaţiei intrării K, se va utiliza regula 2 (obligatorie) şi regulile opţionale 5 şi 6. În cadrul ecuaţiei intrării K, regula 2 generează termeni (produse logice) obligatorii iar regulile 5 şi 6 generează termeni opţionali. Aceşti termeni opţionali se pot reţine sau se pot elimina. Scopul este de a reţine doar acei termeni opţionali care, alăturaţi termenilor obligatorii, conduc la minimizarea ecuaţiei intrării K.

a) Ecuaţia intrării J_A

Pe ieşirea Q_A, aferentă primului bistabil de stare, apare o tranziţie din ”0” în ”1” la tranziţia automatului din starea ST_1 în starea ST_2 (figura 3.28). Pentru ca această tranziţie să se producă este obligatoriu ca intrarea de setare a bistabilului (J_A) să fie pe ”1” pe durata stării ST_1 (regula 1). Opţional, intrarea J_A poate să fie pe ”1” şi în stările ST_2 şi respectiv ST_3 (regula 4 şi respectiv 3). În baza acestui raţionament (efectuat pe graf de tranziţii din figura 3.28) rezultă ecuaţia (3.17):

$$J_A = ST_1 \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) + \boxed{ST_2 \bullet BUSY + ST_2 \bullet \overline{BUSY} + ST_3 \bullet BUSY + ST_3 \bullet \overline{BUSY}} =$$

$$= ST_1 \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) + \boxed{ST_2 + ST_3} \qquad (3.17)$$

Relativ la ecuaţia (3.17), termenii încadraţi în chenarul dreptunghic sunt termeni opţionali (rezultaţi pe baza regulilor opţionale 4 şi respectiv 3).

b) Ecuaţia intrării K_A

Pe ieşirea Q_A, aferentă primului bistabil de stare, apare o tranziţie din ”1” în ”0” la tranziţia automatului din starea ST_3 în starea ST_0 (figura 3.28). Pentru ca această tranziţie să se producă este obligatoriu ca intrarea de resetare a bistabilului (K_A) să fie pe ”1” pe durata stării ST_3 (regula 2). Opţional (regulile 5 şi 6), intrarea K_A poate să fie pe ”1” şi în stările ST_0 (deoarece la trecerea din ST_0 în ST_1, ieşirea Q_A are o tranziţie din ”0” în ”0”) şi respectiv în ST_1 (deoarece la trecerea din ST_1 în ST_2 , ieşirea Q_A are o tranziţie din ”0” în ”1” iar la trecerea din ST_1 în ST_0 , ieşirea Q_A are o tranziţie din ”0” în ”0”). În baza acestui raţionament (efectuat pe graful de tranziţii din figura 3.28) rezultă ecuaţia (3.18):

$$K_A = ST_3 \bullet \overline{BUSY} + \boxed{ST_0 + ST_1 \bullet (\overline{MIR}_{19} + MIR_{19} \bullet \overline{MIR}_{18} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4) + ST_1 \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4)} =$$
$$= ST_3 \bullet \overline{BUSY} + \boxed{ST_0 + ST_1} \qquad (3.18)$$

Relativ la ecuaţia (3.18), termenii încadraţi în chenarul dreptunghic sunt termeni opţionali (rezultaţi pe baza regulilor opţionale 5 şi respectiv 6).

c) Ecuaţia intrării J_B

Pe ieşirea Q_B apare o tranziţie din ”0” în ”1” la tranziţia automatului din starea ST_0 în starea ST_1 (figura 3.28). Pentru ca această tranziţie să se producă este obligatoriu ca intrarea de setare a bistabilului (J_B) să fie pe ”1” pe durata stării ST_0 (regula 1). Opţional, intrarea J_B poate să fie pe ”1” şi în stările ST_1 (deoarece la trecerea din ST_1 în ST_2 , ieşirea Q_B are o tranziţie din ”1” în ”1” iar la trecerea din ST_1 în ST_0 , ieşirea Q_B are o tranziţie din ”1” în ”0”) şi respectiv ST_2 (deoarece la trecerea din ST_2 în ST_2 , ieşirea Q_B are o tranziţie din ”1” în ”1” iar la trecerea din ST_2 în ST_3 , ieşirea Q_B are o tranziţie din ”1” în ”0”). În baza acestui raţionament (efectuat pe graful de tranziţii din figura 3.28) rezultă ecuaţia (3.19):

$$J_B = ST_0 + \boxed{ST_1 \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) + ST_1 \bullet (\overline{MIR}_{19} + MIR_{19} \bullet \overline{MIR}_{18} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4)} +$$
$$+ \boxed{ST_2 \bullet \overline{BUSY} + ST_2 \bullet BUSY} = ST_0 + \boxed{ST_1 + ST_2} \qquad (3.19)$$

Relativ la ecuaţia (3.17), termenii încadraţi în chenare dreptunghice sunt termeni opţionali (rezultaţi pe baza regulilor opţionale 4 şi respectiv 3). Să observăm că, în ecuaţia (3.19), termenii opţionali din primele două chenare dreptunghice, prin minimizare, au generat termenii opţionali din cel de-al treilea chenar dreptunghic. Minimizarea se bazează evident pe graful de tranziţii din figura 3.28.

d) Ecuaţia intrării K_B

Ieşirea Q_B are următoarele tranziţii din ”1” în ”0”:

- La tranziţia automatului din ST_1 în ST_0. Această tranziţie induce în ecuaţia intrării K_B termenul obligatoriu:

$$ST_1 \bullet (\overline{MIR}_{19} + MIR_{19} \bullet \overline{MIR}_{18} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4).$$

- La tranziţia automatului din ST_2 în ST_3. Această tranziţie induce în ecuaţia intrării K_B termenul obligatoriu:

$$ST_2 \bullet \overline{BUSY}$$

Termenii opţionali sunt generaţi de tranziţiile din ”0” în ”0” (regula 5) şi respectiv din ”0” în ”1” (regula 6). Ieşirea Q_B are următoarele tranziţii din ”0” în ”0”:

- La tranziţia automatului din ST_3 în ST_0. Această tranziţie induce în ecuaţia intrării K_B termenul opţional: $ST3 \bullet \overline{BUSY}$.

- La tranziţia automatului din ST_3 în ST_3. Această tranziţie induce în ecuaţia intrării K_B termenul opţional: $ST3 \bullet BUSY$.

Ieşirea Q_B are tranziţie din ”0” în ”1”, la tranziţia automatului din ST_0 în ST_1. Această tranziţie induce în ecuaţia intrării K_B termenul opţional: ST_0.

Ecuaţia (3.20) reprezintă ecuaţia intrării K_B. Aceasta se obţine reunind (funcţia SAU) termenii obligatorii şi respectiv opţionali enumeraţi mai sus.

$$K_B = ST_1 \bullet (\overline{MIR}_{19} + MIR_{19} \bullet \overline{MIR}_{18} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4) + ST_2 \bullet \overline{BUSY} + \boxed{ST_3 \bullet \overline{BUSY} + ST_3 \bullet BUSY + ST_0} =$$

$$= ST_1 \bullet (\overline{MIR}_{19} + MIR_{19} \bullet \overline{MIR}_{18} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4) + ST_2 \bullet \overline{BUSY} + \boxed{ST_3 + ST_0} \qquad (3.20)$$

Relativ la ecuaţia (3.20), termenii încadraţi în chenarul dreptunghic sunt termeni opţionali (rezultaţi pe baza regulilor opţionale 5 şi respectiv 6)**.**

Din tabelul 3.15 rezultă ecuaţiile care redau codificarea stărilor automatului:

$$\begin{aligned} ST_0 &= \overline{Q}_A \bullet \overline{Q}_B \\ ST_1 &= \overline{Q}_A \bullet Q_B \\ ST_2 &= Q_A \bullet Q_B \\ ST_3 &= Q_A \bullet \overline{Q}_B \end{aligned} \qquad (3.21)$$

Înlocuind relaţiile (3.21) în ecuaţiile (3.17), (3.18), (3.19) şi (3.20) se obţine sistemul (3.22). Acest sistem sintetizează ecuaţiile intrărilor automatului SEQ implementat cu bistabili JK şi respectiv cu stări codificate.

În ecuaţiile (3.22) am reţinut atât termenii obligatorii, obţinuţi pe baza regulilor 1 şi 2, cât şi termenii opţionali, obţinuţi pe baza regulilor 3, 4, 5 şi 6. Termenii opţionali (încadraţi în chenare dreptunghice) pot fi menţinuţi în cadrul ecuaţiilor sau pot fi eliminaţi din aceste ecuaţii deoarece aceştia nu modifică funcţionarea automatului; cu sau fără ei, funcţionarea automatului este aceeaşi.

Fiecare ecuaţie va fi analizată. Dacă anumiţi termeni opţionali (din cadrul acesteia) conduc la minimizarea (simplificarea) ei, atunci termenii opţionali respectivi se menţin şi se procedează la minimizarea ecuaţiei respective. Termenii opţionali care nu contribuie la minimizarea ecuaţiei vor fi eliminaţi.

$$
\begin{aligned}
J_A &= \overline{Q}_A \bullet Q_B \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) + \boxed{Q_A \bullet Q_B + Q_A \bullet \overline{Q}_B} \\
K_A &= Q_A \bullet \overline{Q}_B \bullet BUSY + \boxed{\overline{Q}_A \bullet \overline{Q}_B + \overline{Q}_A \bullet Q_B} \\
J_B &= \overline{Q}_A \bullet \overline{Q}_B + \boxed{\overline{Q}_A \bullet Q_B + Q_A \bullet Q_B} \\
K_B &= \overline{Q}_A \bullet Q_B \bullet (\overline{MIR}_{19} + MIR_{19} \bullet \overline{MIR}_{18} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4) + Q_A \bullet Q_B \bullet \overline{BUSY} + \boxed{Q_A \bullet \overline{Q}_B + \overline{Q}_A \bullet \overline{Q}_B}
\end{aligned}
\tag{3.22}
$$

Analizând sistemul (3.22) constatăm că termenii opţionali nu conduc la simplificări în ecuaţiile intrărilor J_A, K_A şi K_B. În consecinţă, vom renunţa la termenii opţionali din cadrul acestor trei ecuaţii.

În schimb, dacă în ecuaţia întrării J_B reţinem doar primul termen opţional şi renunţăm la al doilea, vom obţine o uşoară simplificare (minimizare). Etapele procesului de simplificare sunt redate în cadrul relaţiei (3.23).

$$
J_B = \overline{Q}_A \bullet \overline{Q}_B + \boxed{\underbrace{\overline{Q}_A \bullet Q_B}_{\text{Se reţine}} + \underbrace{Q_A \bullet Q_B}_{\text{Se elimină}}} = \overline{Q}_A \bullet \overline{Q}_B + \overline{Q}_A \bullet Q_B = \overline{Q}_A \bullet (\overline{Q}_B + Q_B) = \overline{Q}_A \tag{3.23}
$$

Ecuaţiile (3.24) redau forma finală a ecuaţiilor intrărilor:

$$
\begin{aligned}
J_A &= \overline{Q}_A \bullet Q_B \bullet MIR_{19} \bullet MIR_{18} \bullet (MIR_5 + MIR_4) = \\
&= \overline{Q}_A \bullet Q_B \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_5 + \overline{Q}_A \bullet Q_B \bullet MIR_{19} \bullet MIR_{18} \bullet MIR_4 \\
K_A &= Q_A \bullet \overline{Q}_B \bullet BUSY \\
J_B &= \overline{Q}_A \\
K_B &= \overline{Q}_A \bullet Q_B \bullet (\overline{MIR}_{19} + MIR_{19} \bullet \overline{MIR}_{18} + MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4) + Q_A \bullet Q_B \bullet \overline{BUSY} = \\
&= \overline{Q}_A \bullet Q_B \bullet \overline{MIR}_{19} + \overline{Q}_A \bullet Q_B \bullet MIR_{19} \bullet \overline{MIR}_{18} + \overline{Q}_A \bullet Q_B \bullet MIR_{19} \bullet MIR_{18} \bullet \overline{MIR}_5 \bullet \overline{MIR}_4 + Q_A \bullet Q_B \bullet \overline{BUSY}
\end{aligned}
\tag{3.24}
$$

C. Ecuaţiile ieşirilor

Ieşirile unui automat reprezintă comenzile pe care acesta le generează. Toate aceste comenzi se regăsesc în organigrama de funcţionare a automatului (figura 3.7). Ecuaţiile (3.14) elaborate la 3.5.6.B. reprezintă ecuaţiile ieşirilor (comenzilor) extrase direct din organigrama de funcţionare a automatului SEQ.

Pe de altă parte, cele patru stări ale automatului SEQ, în varianta de implementare „cu bistabili şi stări codificate”, au fost codificate cu ajutorul a doi bistabili de stare. Modul de codificare a celor patru stări a fost sintetizat prin relaţiile (3.21).

Prin urmare, ecuaţiile ieşirilor aferente automatului SEQ cu stări codificate, se vor obţine prin simpla înlocuire a relaţiilor (3.21) în ecuaţiile (3.14). Relaţiile (3.25) redau rezultatul acestei înlocuiri.

$$
\begin{aligned}
&LdMIR=\overline{Q}_A\bullet\overline{Q}_B\\
&+1MAR=\overline{Q}_A\bullet Q_B\bullet\overline{MIR}_{19}\bullet\overline{g}+\overline{Q}_A\bullet Q_B\bullet MIR_{19}\\
&LdMAR=\overline{Q}_A\bullet Q_B\bullet\overline{MIR}_{19}\bullet g\\
&EN_1=\overline{Q}_A\bullet Q_B\bullet MIR_{19}\\
&MREQ=Q_A\bullet Q_B\bullet\overline{BUSY}+Q_A\bullet\overline{Q}_B\\
&WR=EN_3=Q_A\bullet\overline{Q}_B\bullet MIR_5\bullet MIR_4\\
&RD=PdMEM=Q_A\bullet\overline{Q}_B\bullet\overline{MIR_5\bullet MIR_4}\\
&PmM_{SEQ}=Q_A\bullet\overline{Q}_B\bullet MIR_5\bullet\overline{MIR}_4\bullet\overline{BUSY}\\
&PmIR_{SEQ}=Q_A\bullet\overline{Q}_B\bullet\overline{MIR}_5\bullet MIR_4\bullet\overline{BUSY}
\end{aligned}
\qquad (3.25)
$$

D. Schema detaliată a automatului

Schema detaliată a automatului SEQ cu stări codificate se obţine într-o manieră similară cu cea a automatului cu stări necodificate (vezi paragraful 3.5.6.C.). Procedura de urmat conţine trei paşi:

a) se implementează în *hardware* ecuaţiile intrărilor (3.24)

b) se implementeză în *hardware* ecuaţiile ieşirilor (3.25)

c) se adaugă implementările *hardware* obţinute în paşii anteriori pe structura de bază a automatului (figura 3.29)

Rezultatul obţinut după parcurgerea acestor trei paşi (figura 3.30) reprezintă schema detaliată a automatului SEQ cu stări codificate şi bistabili JK *master-slave*.

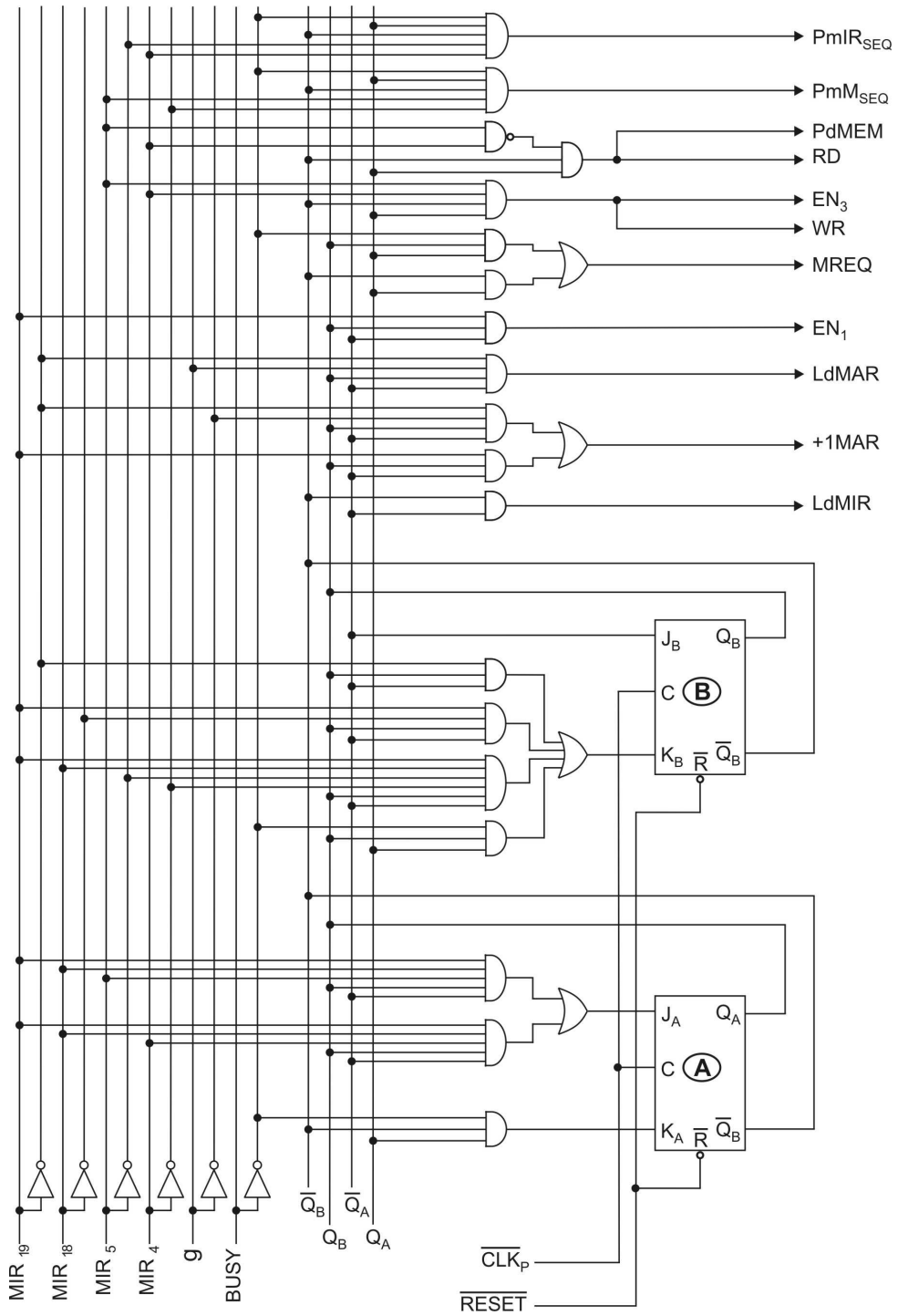


Fig 3.30 Automatul SEQ implementat cu bistabili JK şi stări codificate

3.6 Schema bloc a unităţii de control microprogramate aferentă procesorului didactic

Pornind de la schema bloc de principiu a unităţii de control microprogramate (figurile 3.1 şi 3.4), pe baza formatului microinstrucţiunii stabilit la 3.4 şi a automatului SEQ proiectat la 3.5, se poate elabora schema bloc a unităţii de control microprogramate aferentă procesorului didactic (figura 3.31).

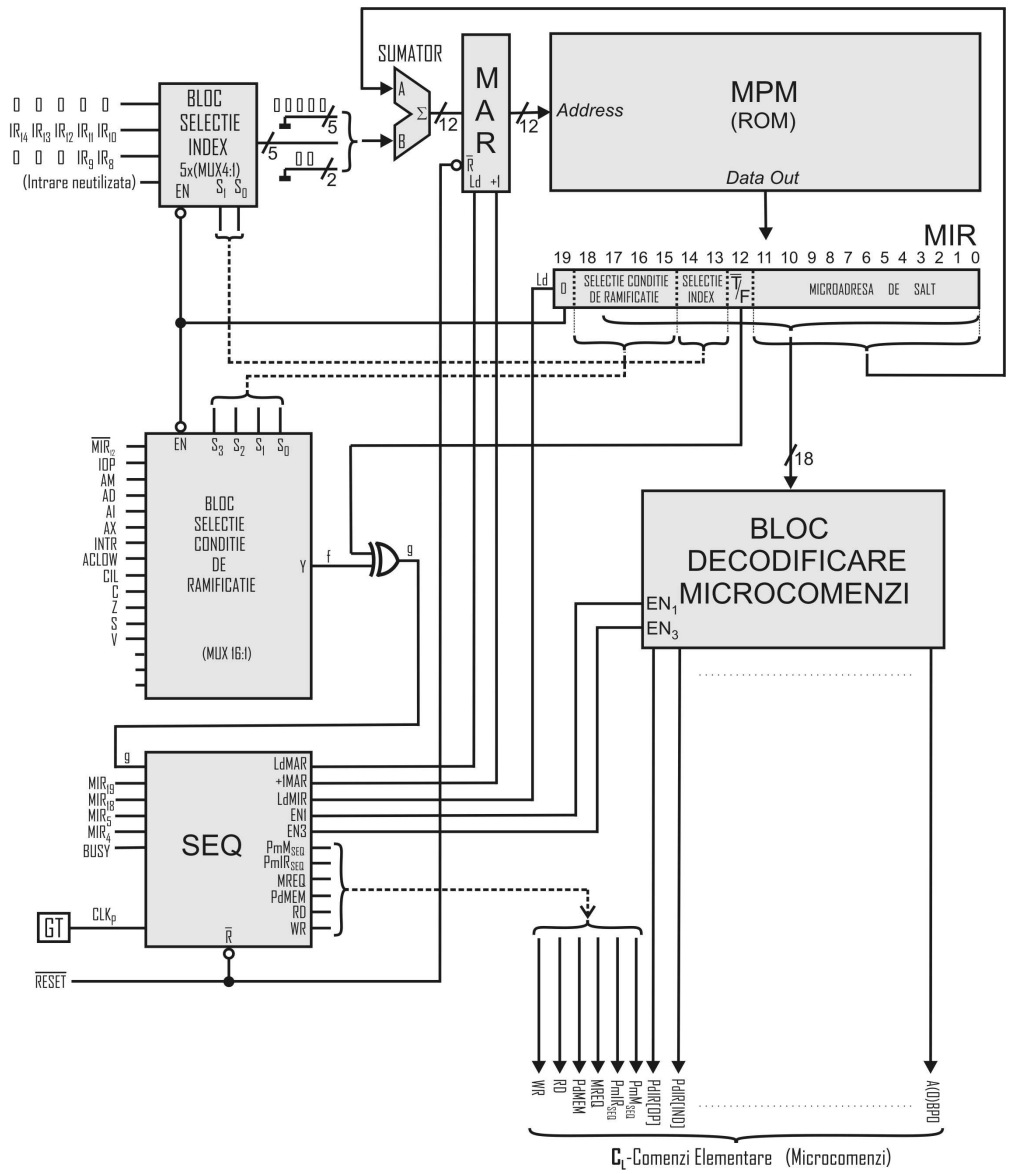


Fig 3.31 Schema bloc a unităţii de control microprogramate aferentă procesdorului didactic

În structura unităţii de control din figura 3.31 regăsim următoarele blocuri componenete:

MIR -***M****icro****i****nstruction* ***R****egister* (registrul microinstrucţiunii). În MIR se încarcă microinstrucţiunea selectată în MPM de microadresa prezentă în MAR (*fetch*-ul microinstrucţiunii). Pentru încărcarea microinstrucţiunii în MIR, în starea ST_0, automatul SEQ activează comanda LdMIR (figura 3.7).

MAR -***M****icro****a****ddress* ***R****egister* (registrul de microadrese). MAR joacă rol de *microprogram counter* (adresează în MPM microinstrucţiunea ce urmează a fi citită şi încărcată în MIR (*fetch*-ul microinstrucţiunii). Pentru încărcarea în MAR a microadresei următoarei microinstrucţiuni SEQ utilizează două comenzi: LdMAR (pentru succesorii JUMP şi JUMPI) şi respectiv +1MAR (pentru succesorul STEP).

MPM -***M****icro****p****rogram* ***M****emory* (memoria de microprogram). MPM conţine microprogramul de emulare a setului de instrucţiuni aferent procesorului didactic. Microprogramul este format dintr-o colecţie de microrutine: microrutina *fetch* instrucţiune, microrutine *fetch* operand (câte una pentru fiecare mod de adresare), microrutine de execuţie (câte una pentru fiecare instrucţiune) şi microrutina de întrerupere.

SEQ -***Seq****uencer* (automat secvenţiator). SEQ generează comenzile interne prin care controlează *fetch*-ul şi execuţia microinstrucţiunilor; a fost proiectat la 3.5.

BLOC SELECŢIE CONDIŢIE DE RAMIFICAŢIE –genereză funcţia globală de ramificaţie g. Acest bloc are rol în execuţia succesorilor condiţionaţi din cadrul microinstrucţiunilor de ramificaţie (tabelele 3.1 şi respectiv 3.2). Prin urmare, blocul este validat cu $MIR_{19}=0$ (dacă în MIR se află o microinstrucţiune de ramificaţie). În cadrul acestor microinstrucţiuni se selectează o condiţie care este testată. Succesorii condiţionaţi au forma generică:

SUCCESOR1 else SUCCESOR2

Dacă condiţia testată (pe fals sau pe adevărat) este îndeplinită, atunci BLOCUL DE SELECŢIE CONDIŢIE DE RAMIFICAŢIE generează $g=1$. În consecinţă, se execută SUCCESOR1. Dacă condiţia testată este neândeplinită, atunci BLOCUL DE SELECŢIE CONDIŢIE DE RAMIFICAŢIE generează $g=0$. În consecinţă, se execută SUCCESOR2.
Pentru toate cele 12 microinstrucţiuni în discuţie (tabelele 3.1 şi respectiv 3.2), SUCCESOR1 este JUMPI µADR+INDEX (respectiv JUMP µADR dacă INDEX-ul selectat are valoarea zero), iar SUCCESOR2 este STEP.

După cum s-a arătat la 3.4.1.B, execuţia succesorului de salt indexat JUMPI µADR+INDEX constă în operaţa:

$$MAR \leftarrow MIR_{11\div0} + INDEX \mid_{\text{selectat cu } MIR_{14,13}}$$

În acord cu cele prezentate la 3.4.1.A, execuţia succesorului JUMP μADR constă în:

$$MAR \leftarrow MIR_{11\div0}$$

iar, execuţia succesorului STEP constă în operaţia:

$$MAR \leftarrow MAR+1$$

Funcţia g este testată de către automatul SEQ. Dacă g=1, SEQ va activa comanda LdMAR (necesară execuţiei succesorilor JUMPI μADR+INDEX şi respectiv JUMP μADR). Dacă *g*=0 (dacă condiţia testată este neândeplinită), SEQ va activa comanda +1MAR (necesară execuţiei succesorului STEP); se va trece astfel secvenţial la următoarea microinstrucţiune.

BLOC SELECŢIE INDEX –este format din 5 multiplexoare 4:1 şi generează cei trei indecşi definiţi la 3.4.1 (tabelul 3.3). Are deci rol în execuţia succesorilor JUMPI şi respectiv JUMP. Este validat cu MIR_{19}=0 (dacă în MIR se află o microinstrucţiune de ramificaţie).

Biţii MIR_{14}, MIR_{13} aplicaţi pe intrările de selecţie (S_1 şi respectiv S_0) ale celor 5 multiplexoare 4:1, selectează cei trei indecşi definiţi la 3.4.1 (tabelul 3.3). Pe primele trei intrări aferente celor 5 multiplexoare 4:1 sunt aplicaţi cei trei indecşi definiţi în tabelul 3.3 ($INDEX_1$, $INDEX_2$ şi respectiv $INDEX_3$); Pentru că cel de-al patrulea index ($INDEX_4$) este nedefinit în tabelul 3.3, ultima intrare în cele 5 multiplexoare (intrarea 3) este neutilizată.

<u>*Observaţie*</u>:
Indecşii sunt codificaţi pe 12 biţi. Analizând cei trei indecşi definiţi în tabelul 3.3, constatăm:

a) cei mai puţin semnificativi doi biţi au valoarea zero în cei trei indecşi definiţi ($INDEX_1$, $INDEX_2$ şi $INDEX_3$).

b) cei mai semnificativi cinci biţi au valoarea zero în cei trei indecşi definiţi.

c) cei cinci biţi mediani (biţii rămaşi) iau valori diferite pentru indecşi diferiţi (sunt zero doar la $INDEX_1$). Prin urmare, BLOCUL DE SELECŢIE INDEX va opera doar pe aceşti ultimi cinci biţi; ceilalţi şapte vor fi generaţi în permanenţă pe zero. Se realizează astfel o simplificare structurală a BLOCUL-lui de SELECŢIE INDEX.

SUMATOR -sumator pe 12 biţi cu rol în execuţia succesorilor JUMPI şi JUMP. La JUMPI, INDEX-ul slectat de câmpul $MIR_{14,13}$ trebuie adunat la câmpul MICROADRESA DE SALT preluat din microinstrucţiunea curentă ($MIR_{11\div0}$). SUMATOR-ul relizează această operaţie de adunare. Dacă INDEX-ul selectat are valoarea zero, succesorul JUMPI se transformă în JUMP.

BLOC DECODIFICARE MICROCOMENZI -generează comenzile logice elementare C_L (microcomenzile) codificate în microinstrucţiunea curentă. La 3.4 am arătat că doar microinstrucţiunile de transfer (sincron şi respectiv asincron) conţin câmpuri în care sunt codificate microcomenzi. Aceste microcomenzi trebuie activate în starea ST_1 a automatului SEQ (starea de execuţie a microinstrucţiunii). Acest lucru se realizează cu ajutorul comenzii de validare EN_1, activată de către SEQ în starea ST_1.

Reamintim faptul că, în microinstrucţiunea de transfer asincron există patru câmpuri (SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS) care au semnificaţie numai dacă în câmpul MEMORY OPERATION este codificată microcomanda WRITE şi care au rolul de a emite pe RBUS (MBUS) datele de scris în memorie. Scrierea în memorie se face în starea ST_3 a automatului SEQ, iar validarea microcomenzilor codificate în cele patru câmpuri menţionate se face cu ajutorul comenzii de validare EN_3, pe care SEQ o activează în starea ST_3 (figura 3.7).

O microinstrucţiune este procesată în două perioade de tact succesive. Pentru automatul SEQ, cele două perioade de tact înseamnă două stări: ST_0 şi respectiv ST_1. În starea ST_0 se execută *fetch*-ul microinstrucţiunii (microinstrucţiunea adresată de MAR în MPM, este încărcată în MIR cu ajutorul comenzii LdMIR). Starea ST_1 va fi starea de execuţie a microinstrucţiunii. Prin urmare, se activează microcomenzile codificate în microinstrucţiune prin validarea decodificatoarelor de microcomenzi din cadrul BLOC-ului de DECODIFICARE MICROCOMENZI. Validarea decodificatoarelor este realizată de către SEQ prin comanda EN_1. Tot în starea ST_1 se va hotărâ succesoarea microinstrucţiunii curente (se încarcă în MAR microadresa următoarei microinstrucţiuni). Fixarea în MAR a microadresei următoarei microinstrucţiuni de procesat, se face în funcţie de succesorul codificat în microinstrucţiunea curentă:

- Pentru succesorul STEP, SEQ activează comanda +1MAR (se execută operaţia MAR←MAR+1).
- Pentru succesorul JUMPI, SEQ activează comanda LdMAR (se execută operaţia $MAR \leftarrow MIR_{11\div0} + INDEX|_{\text{selectat cu } MIR_{14,13}}$).
- Pentru succesorul JUMP, SEQ activează comanda LdMAR (ca la JUMPI) dar va fi selectat un index egal cu zero ($INDEX_1$ din tabelul 3.3). Se execută deci operaţia $MAR \leftarrow MIR_{11\div0}$.

Reamintim faptul că, în cazul microinstrucţiunilor de transfer asincron care lansează efectiv o operaţie de citire din sau de scriere în memorie, SEQ parcurge patru stări. În primele două stări (ST_0 şi respectiv ST_1), SEQ execută operaţiile descrise mai sus. În următoarele două stări (ST_2 şi respectiv ST_3), SEQ controlează operaţia de transfer procesor-memorie.

3.7 Proiectarea blocurilor din componenţa unităţii de control microprogramate

În paragraful anterior au fost introdusă schema bloc a unităţii de control microprogramate (figura 3.31) şi au fost descris succint rolul fiecărui bloc din componenţa acestei unităţi. În cadrul prezentului paragraf vom trece de la descrierea succintă la proiectarea detaliată a fiecărui bloc din componenţa unităţi de control. Remintim faptul că proiectarea automatului SEQ a fost deja realizată (paragraful 3.5). Vom continua, prin urmare, cu celelalte blocuri din structura unităţii microprogramate.

3.7.1 Blocul de decodificare microcomenzi

Microcomenzile (C_L) se generează prin decodificarea câmpurilor microinstrucţiunilor de transfer (sincron şi asincron). Microcomenzile generate din microinstrucţiunea de transfer sincron sunt activate în starea ST_1 a automatului SEQ. Comanda pe care am prevăzut-o pentru validarea (activarea) acestor microcomenzi este EN_1 (vezi organigrama SEQ descrisă la 3.5.1). Microinstrucţiunea de transfer asincron generează microcomenzi în stările ST_1 (comenzile codificate în câmpul SHIFT & OTHER OPERATIONS) şi respectiv ST_3 (comenzile codificate în cele patru câmpuri: SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS, câmpuri care au semnificaţie numai dacă în câmpul MEMORY OPERATION este codificată microcomanda WRITE). Pentru validarea (activarea) microcomenzilor pe durata stării ST_3 a automatului am prevăzut comanda EN_3 (vezi organigrama SEQ descrisă la 3.5.1). La implementarea BLOCULUI DE DECODIFICARE MICROCOMENZI trebuie ţinut cont de ambele surse de generare. Remintim faptul că şi automatul SEQ, în stările ST_2 şi ST_3, activează în mod direct câteva microcomenzi prin care controlează transferul procesor-memorie (vezi figurile 3.11 şi 3.7 cu descrierile aferente).

A. Generarea microcomenzilor pentru selecţia sursei pe SBUS, DBUS, RBUS şi pentru ALU

Microcomenzile pentru selecţia sursei pe SBUS, DBUS, RBUS şi pentru ALU sunt codificate în microinstrucţiunile de transfer sincron şi respectiv asincron, în câmpurile: SURSA SBUS, SURSA DBUS, SURSA RBUS şi OPERAŢIE ALU (vezi figurile 3.3 şi 3.4 şi tabelele 3.4, 3.5, 3.6 şi 3.7).
Toate aceste microcomenzi sunt de tip nivel şi se generează din două surse:

a) din microinstrucţiunea de transfer sincron, în starea ST_1 a automatului SEQ.

b) din microinstrucţiunea de transfer asincron, în starea ST_3 a automatului SEQ, dacă în câmpul MEMORY OPERATION din această microinstrucţiune este codificată

microcomanda WRITE ($MIR_{5,4}$=1,1). În acest caz, comenzile în discuţie vor emite pe RBUS datele de scris în memorie (vezi paragraful C3.3 de la 3.5.1).

Pentru decodificarea celor patru câmpuri (SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS) vom utiliza decodificatoare 3:8 (figura 3.32). Cele patru decodificatoare decodifică , în ordine, cele patru câmpuri menţionate. Validarea celor patru decodificatoare este realizată cu semnalul:

$$EN=EN_1+EN_3 \qquad (3.26)$$

Ecuaţia (3.26) reuneşte cele două surse de generare descrise mai sus; EN_1 validează sursa a) iar EN_3 validează sursa b). Comenzile EN_1 şi EN_3 sunt activate de automatul SEQ, în stările ST_1 şi respectiv ST_3 (vezi 3.5.1).

Observaţii:

1. *Validarea EN_1 operează (în starea ST_1 a automatului SEQ) atât pentru microinstrucţiunea de transfer sincron cât şi pentru cea de transfer asincron (vezi 3.5.1). Validarea EN_3 operează (în starea ST_3 a automatului SEQ, numai pentru microinstrucţiunea de transfer asincron şi numai dacă în această microinstrucţiune este codificată microcomanda WRITE. Prin urmare, în cazul unei microinstrucţiuni de transfer asincron, comenzile codificate în cele patru câmpuri sunt activate de două ori succesiv: mai întâi în starea ST_1 a automatului SEQ şi apoi în starea ST_3, când trebuie emise pe RBUS (MBUS) datele de scris în memorie. Validarea în ST_1 nu este necesară dar nu deranjează. Noi chiar vom exploata această succesiune de două validări în microrutinele de execuţie aferente instrucţiunilor INC src şi respectiv DEC src, plasate la etichetele INC şi respectiv DEC în cadrul microprogramului de emulere (vezi 3.8.2). În cadrul celor două microrutine, printr-o singură microinstrucţiune de transfer asincron, care conţine microcomenzile WRITE (codificată în câmpul MEMORY OPERATION) şi respectiv PdCOND (codificată în câmpul SHIFT& OTHER OPERATIONS), vom reuşi să scriem în memorie (microcomanda WRITE) valoarea src+1 (la instrucţiunea INC), respectiv src-1 (la instrucţiunea DEC) şi să poziţionăm flag-urile de condiţii (microcomanda PdCOND) în acord cu rezultatul încrementării; PdCOND operează în starea ST_1 a automatului SEQ iar WRITE în starea ST_3.*

2. *Câmpurile SURSA SBUS, SURSA DBUS, SURSA RBUS şi OPERAŢIE ALU ocupă aceleaşi poziţii binare atât în microinstrucţiunea de transfer sincron cât şi în cea de transfer asincron. Această suprapunere a avut drept scop simplificarea implementării hardware a decodificatoarelor aferente celor patru câmpuri (figura 3.32).*

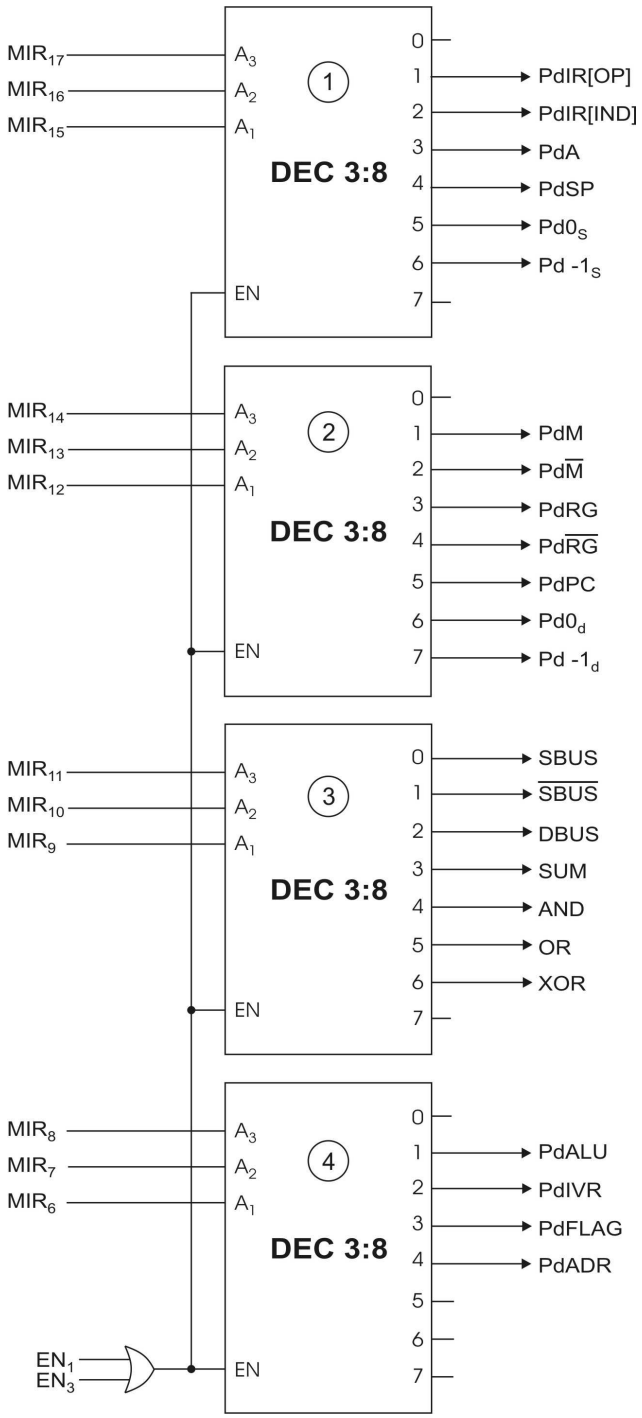


Fig 3.32 Generarea microcomenzilor pentru selecţia sursei pe SBUS, DBUS, RBUS şi a microcomenzilor pentru unitatea ALU

B. Generarea microcomenzilor „*Primeşte Registru*”

Generarea microcomenzilor „*Primeşte Registru*” se face prin decodificarea câmpului $MIR_{5\div3}$ din microinstrucţiunea de transfer sincron (câmpul DESTINAŢIE RBUS). Acest câmp codifică 7 destinaţii de tip registru şi o destinaţie NONE (nicăieri). Dat fiind faptul că $MIR_{5\div3}$ este un câmp de trei biţi, pentru decodificare vom utiliza un decodificator 3:8 (figura 3.33).

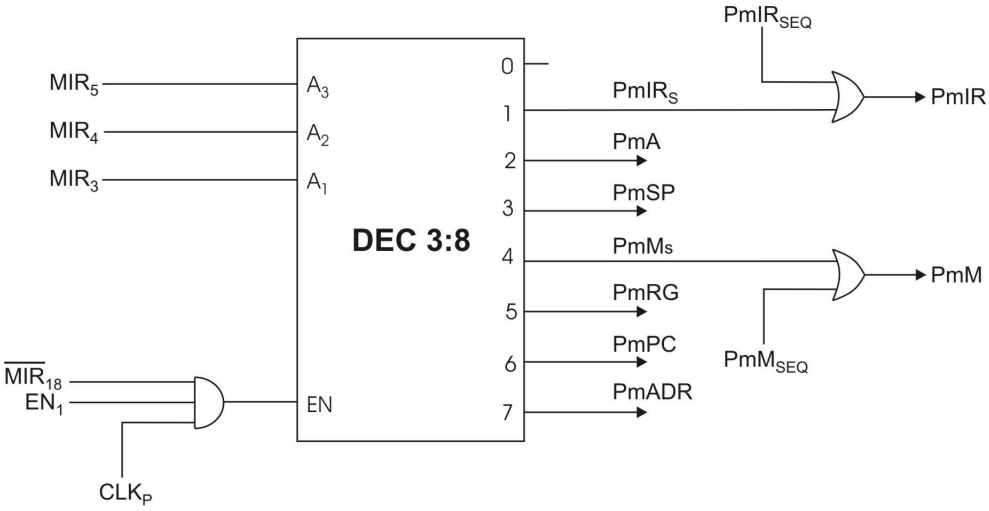


Fig 3.33 Generarea microcomenzilor „*Primeşte Registru*”

Asignarea intrărilor şi respectiv ieşirilor din decodificator s-a făcut în acord cu formatul câmpului DESTINAŢIE RBUS din microinstrucţiunea de transfer sincron (tabelul 3.8).

Validarea decodificatorului se realizează cu semnalul:

$$EN=\overline{MIR_{18}}\bullet EN_1\bullet CLK_P \qquad (3.27)$$

În conformitate cu ecuaţia (3.27), decodificatorul este validat dacă sunt îndeplinite (concomitent) următoarele trei condiţii:

1 $\overline{MIR_{18}}=1$ (în MIR se află o microinstrucţiune de transfer sincron)

2 $EN_1=1$ (este activă comanda de validare EN_1 pe care automatul SEQ o activează în starea ST_1)

3 $CLK_P=1$ (doar pe durata impulsului de tact din cadrul stării ST_1)

Observaţie:

Prima condiţie de validare ar putea fi catalogată incompletă deoarece $\overline{MIR_{18}}=1$ *exclude microinstrucţiunea de transfer asincron dar nu o exclude şi pe cea de ramificaţie. Condiţia care ar exclude şi microinstrucţiunea de ramificaţie ar fi:* $MIR_{19} \bullet \overline{MIR_{18}}=1$ (*vezi figura 3.3 cu explicaţiile relative la biţii* MIR_{19} *şi* MIR_{18}). *Totuşi, dacă combinăm primele două condiţii de validare, obţinem condiţia compusă:* $\overline{MIR_{18}} \bullet EN_1=1$. *Având în vedere că SEQ activează* EN_1 *doar în starea de execuţie* (ST_1) *a unei microinstrucţiuni de transfer sincron sau asincron (vezi figura 3.7), condiţia compusă* ($\overline{MIR_{18}} \bullet EN_1=1$), *exprimată în cuvinte, înseamnă: stare de execuţie a unei microinstrucţiuni de transfer sincron. Prin urmare, includerea condiţiei* $EN_1=1$ *în ecuaţia semnalului de validare, permite trecerea de la varianta completă a primei condiţii* ($MIR_{19} \bullet \overline{MIR_{18}}=1$) *la varianta simplificată* ($\overline{MIR_{18}}=1$). *Altfel spus, prin includerea condiţiei* $EN_1=1$ *în ecuaţia semnalului de validare, devine posibilă o uşoară minimizare a acestui semnal de validare (minimizarea este întotdeauna utilă deoarece conduce la simplificări în hardware).*

Cea de a treia condiţie ($CLK_P=1$) este o condiţie suplimentară care face ca pe ieşirile decodificatorului să se obţină comenzi de tip impuls şi nu de tip nivel. În paragraful 2.5.B. am evidenţiat utilitatea comenzilor de tip impuls şi respectiv de tip nivel şi am arătat de ce comenzile „*primeşte registru*” trebuie să fie comenzi de tip impuls (vezi figura 2.18 cu explicaţiile aferente).

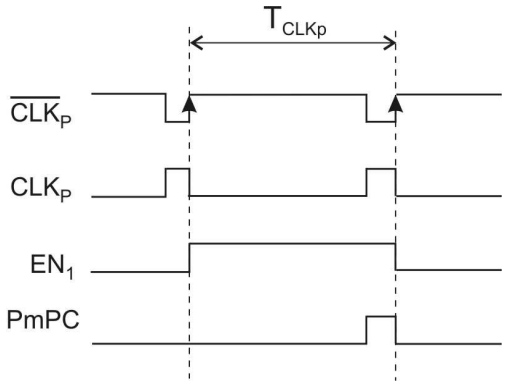


Fig 3.34 Generarea microcomenzilor de tip impuls
(exemplificare pe comanda PmPC)

Includerea variabilei CLK_P în ecuaţia semnalului de validare are ca efect deci, generarea unor comenzi de tip impuls pe ieşirile decodificatorului din figura 3.33; fără variabila CLK_P în ecuaţia (3.10) s-ar fi obţinut comenzi de tip nivel. Forma comenzilor de tip impuls obţinute pe ieşirile decodificatorului din figura 3.33 este redată în figura 3.34.

Pe de altă parte, la 3.5.1 am evidenţiat cele două surse de generare aferente comenzilor PmIR şi respectiv PmM (vezi ecuaţiile (3.7) cu explicaţiile aferente). Ecuaţiile (3.7) sunt implementate în *hardware* prin intermediul celor două porţi SAU din figura 3.33.

C. Generarea microcomenzilor *„Other Operations”*

Microcomenzile *„Other Operations”* sunt codificate în câmpul $MIR_{2\div0}$ din microinstrucţiunea de transfer sincron (vezi figura 3.3 şi tabelul 3.9). Prin urmare, aceste microcomenzi vor putea fi generate tot cu ajutorul unui decodificator 3:8 (figura 3.35). Validarea decodificatorului este realizată cu semnalul:

$$EN=\overline{MIR_{18}}\bullet EN_1 \qquad (3.28)$$

Prin urmare, decodificatorul este validat în starea de execuţie (ST_1) a unei microinstrucţiuni de transfer sincron (vezi observaţia de la 3.7.1.B). Toate microcomenzile din categoria *„Other Operations”* pot fi de tip nivel, cu excepţia microcomenzii PmFLAG. Schema de generare din figura 3.35 se conformează acestei decizii; singura comandă de tip impuls este PmFLAG* care se obţine pe baza comenzii de tip nivel PmFLAG şi a semnalului de tact aferent procesorului CLK_P.

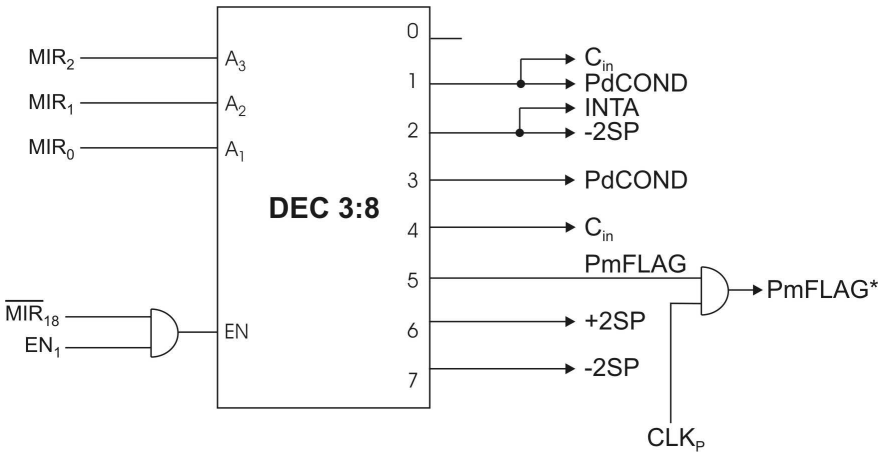


Fig 3.35 Generarea microcomenzilor *„Other Operations”*

D. Generarea microcomenzilor *„Shift&Other Operations”*

Microcomenzile *„Shift&Other Operations”* sunt codificate în câmpul $MIR_{3\div0}$ din microinstrucţiunea de transfer asincron (vezi figura 3.4 şi tabelul 3.11). Fiind codificate într-un câmp de patru biţi ($MIR_{3\div0}$), microcomenzile *„Shift&Other Operations”* vor fi generate cu ajutorul unui decodificator 4:16 (figura 3.36).

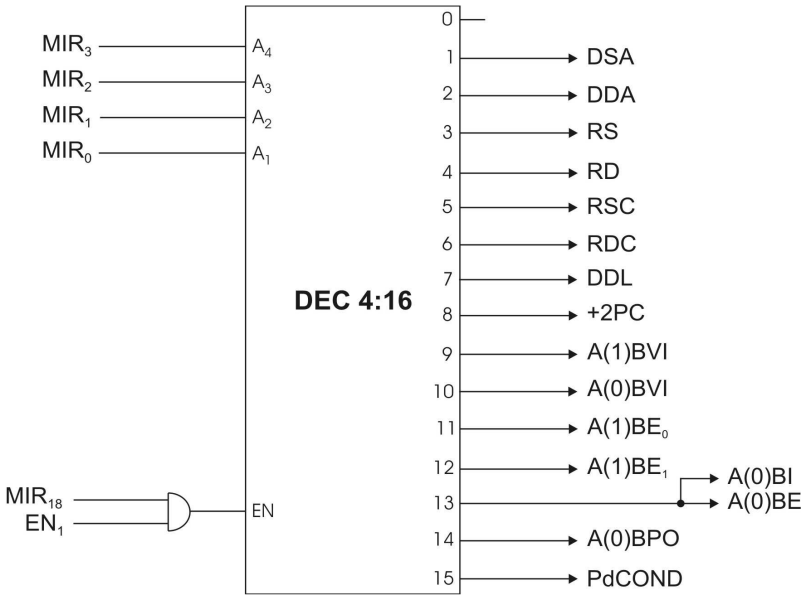


Fig 3.36 Generarea microcomenzilor *„Shift&Other Operations”*

Validarea decodificatorului este realizată cu semnalul:

$$EN=MIR_{18} \bullet EN_1 \qquad (3.29)$$

Prin urmare, decodificatorul este validat în starea de execuţie (ST_1) a unei microinstrucţiuni de transfer asincron (spre deosebire de decodificatorul din figura 3.35 care este validat în starea de execuţie a unei microinstrucţiuni de transfer sincron).

<u>Observaţie</u>: Câmpul MEMORY OPERATION ($MIR_{5,4}$) din cadrul microinstrucţiunii de transfer asincron este decodificat (testat) direct de către automatul SEQ care, în stările ST_2 şi ST_3, iniţiază acţiuni dependente de microcomenzile codificate în acest câmp (vezi paragraful C3. de la 3.5.1).

3.7.2 Blocul de selecţie index

BLOCUL DE SELECŢIE INDEX intervine în execuţia succesorilor JUMPI şi respectiv JUMP având rolul de a genera cei trei indecşi definiţi la 3.4.1 (tabelul 3.3). Deşi indecşii definiţi în tabelul 3.3 ($INDEX_1$, $INDEX_2$ şi $INDEX_3$) sunt pe 12 biţi, blocul de selecţie index poate fi implementat cu doar 5 multiplexoare 4:1 (figura 3.37). Această simplificare este posibilă deoarece cei trei indecşi definiţi în tabelul 3.3 diferă doar pe zona biţilor mediani ($IND_6 \div IND_2$). Biţii IND_{11}, IND_{10}, IND_9, IND_8, IND_7, IND_1 şi IND_0 au valoarea zero în toţi indecşii definiţi în tabelul 3.3 (vezi observaţia de la 3.6).

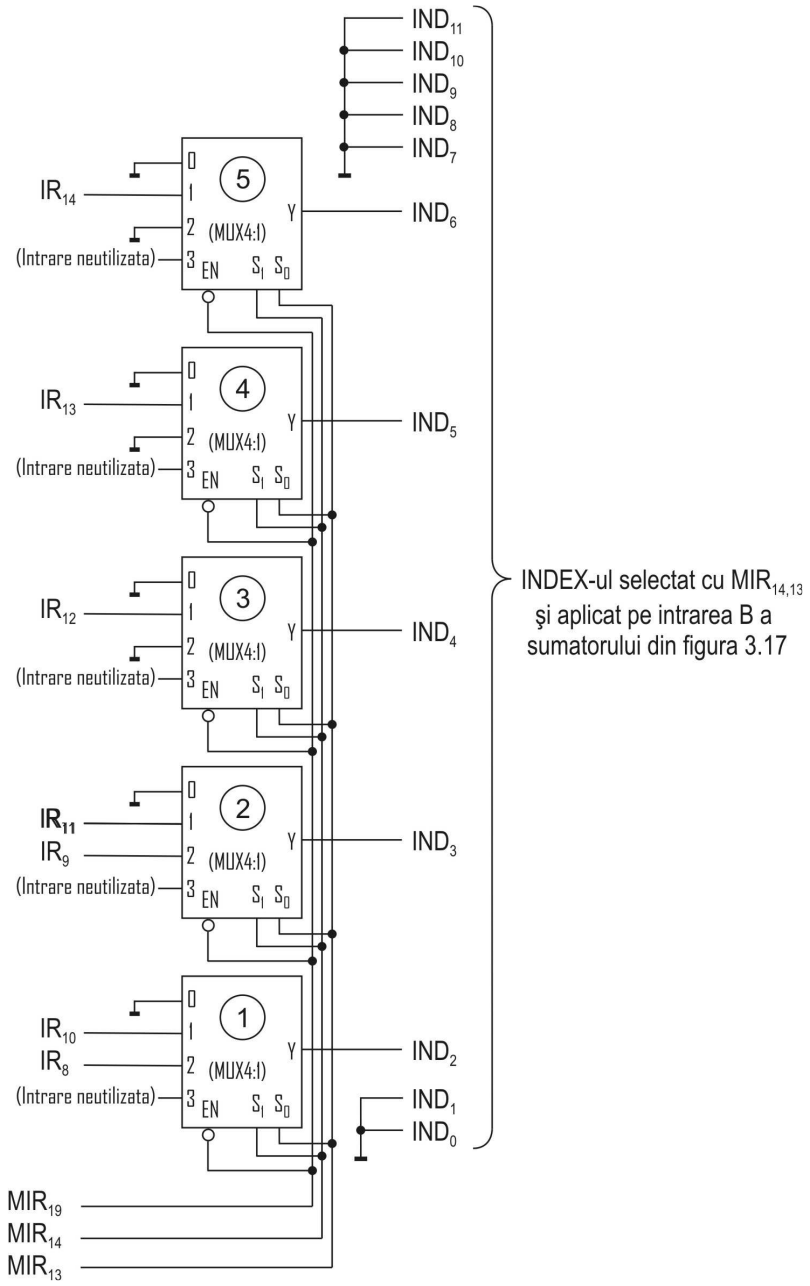


Fig 3.37 Blocul de selecţie index

Succesorii JUMPI şi JUMP pot fi codificaţi doar în microinstrucţiunile de ramificaţie (vezi tabelele 3.1 şi 3.2 de la 3.4.1). Prin urmare, blocul de selecţie index trebuie validat cu MIR_{19}=0 (dacă în MIR se află o microinstrucţiune de ramificaţie). Din acest motiv, intrarea

de validare $\overline{EN}$ (activă în ”0”) aferentă celor 5 multiplexoare din figura 3.37 este acţionată de bitul MIR_{19}.

Cei doi biţi ai câmpului SELECŢIE INDEX (biţii MIR_{14}, MIR_{13}), aplicaţi pe intrările de selecţie (S_1 şi respectiv S_0) ale celor 5 multiplexoare 4:1, selectează cei trei indecşi definiţi la 3.4.1 (tabelul 3.3). Pe primele trei intrări aferente celor 5 multiplexoare 4:1 sunt aplicaţi biţii corespondenţi din cei trei indecşi definiţi în tabelul 3.3 ($INDEX_1$, $INDEX_2$ şi respectiv $INDEX_3$); Pentru că cel de-al patrulea index ($INDEX_4$) este nedefinit în tabelul 3.3, ultima intrare în cele 5 multiplexoare (intrarea 3) este neutilizată.

3.7.3 Blocul de selecţie a condiţiei de ramificaţie

Blocul de selecţie a condiţiei de ramificaţie (figura 3.38) genereză funcţia globală de ramificaţie **g**.

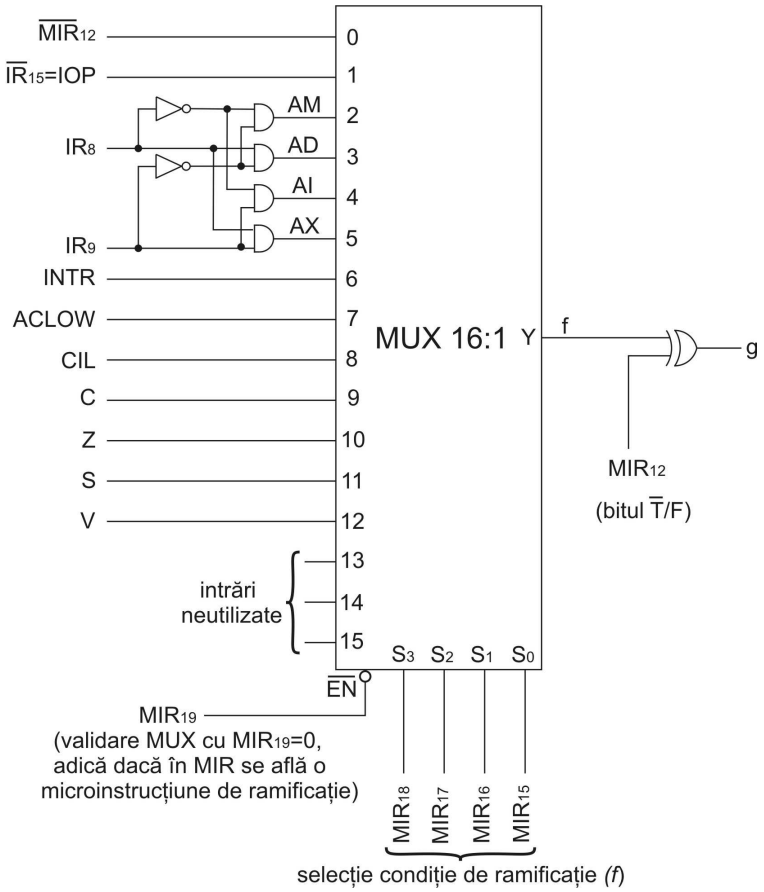


Fig 3.38 Blocul de selecţie a condiţiei de ramificaţie

Acest bloc are rol în execuţia succesorilor condiţionaţi din cadrul microinstrucţiunilor de ramificaţie (tabelele 3.1 şi respectiv 3.2). Prin urmare, blocul este validat cu MIR_{19}=0 (dacă în MIR se află o microinstrucţiune de ramificaţie).

Multiplexorul 16:1 din figura 3.38 generează funcţia de ramificaţie **f** (selectând întocmai condiţiile cuprinse în tabelele 3.1 şi respectiv 3.2). Pe baza funcţiei **f** şi a bitului MIR_{12} ($\overline{T}/F$) se generează funcţia globală de ramificaţie **g**=$f \oplus MIR_{12}$ (vezi 3.4.1). De altfel, tabelele 3.1 şi 3.2 sunt identice din punctul de vedere al condiţiilor testate; diferenţa între cele două tabele o face INDEX-ul (care are valoarea zero în cazul tabelului 3.1 şi respectiv diferită de zero în cazul tabelului 3.2)

În tabelele 3.1 şi 3.2, în afară de primul succesor, toţi ceilalţi succesori sunt condiţionaţi. Succesorii condiţionaţi au forma generică:

SUCCESOR1 else SUCCESOR2

Dacă condiţia testată (pe fals sau pe adevărat) este îndeplinită, atunci BLOCUL DE SELECŢIE CONDIŢIE DE RAMIFICAŢIE generează *g*=1. În consecinţă, se execută SUCCESOR1. Dacă condiţia testată este neîndeplinită, atunci BLOCUL DE SELECŢIE CONDIŢIE DE RAMIFICAŢIE generează *g*=0. În consecinţă, se execută SUCCESOR2.

Pentru toate cele 12 microinstrucţiuni în discuţie (tabelele 3.1 şi respectiv 3.2), SUCCESOR1 este JUMPI µADR+INDEX (respectiv JUMP µADR dacă INDEX-ul selectat are valoarea zero), iar SUCCESOR2 este STEP.

După cum s-a arătat la 3.4.1.B, execuţia succesorului de salt indexat JUMPI µADR+INDEX se realizează prin operaţia:

$$MAR \leftarrow MIR_{11\div0} + INDEX \big|_{\text{selectat cu } MIR_{14,13}}$$

În acord cu cele prezentate la 3.4.1.A, execuţia succesorului JUMP µADR se realizează prin operaţia:

$$MAR \leftarrow MIR_{11\div0}$$

iar, execuţia succesorului STEP se realizează prin operaţia:

$$MAR \leftarrow MAR+1$$

Funcţia g este testată de către automatul SEQ (vezi figurile 3.7 şi 3.17). Dacă g=1, SEQ va activa comanda LdMAR (necesară execuţiei succesorilor JUMPI µADR+INDEX şi respectiv JUMP µADR). Dacă *g*=0 (dacă condiţia testată este neîndeplinită), SEQ va activa comanda +1MAR (necesară execuţiei succesorului STEP); se va trece astfel secvenţial la următoarea microinstrucţiune.

<u>Observaţie</u>:
Primul succesor din tabelele 3.1 şi respectiv 3.2 este necondiţionat (JUMP µADR în tabelul 3.1 şi respectiv JUMPI µADR+INDEX în tabelul 3.2). Pentru ca aceste salturi (necondiţionate!) să fie executate corect, SEQ trebuie să încarce microadresa de salt în MAR, prin activarea comenzii LdMAR. În conformitate cu organigrama din figura 3.7, SEQ activează comanda LdMAR numai dacă funcţia globală de ramificaţie g are valoarea 1. Prin urmare, pentru aceşti succesori de salt necondiţionat, în figura 3.38 se generează (se forţează) g=1, printr-un mic artificiu care poate fi rezumat astfel:

-Cei doi succesori necondiţionaţi (primii succesori din tabelele 3.1 şi 3.2) sunt selectaţi cu $MIR_{18\div15}=0000_2$

-Dacă $MIR_{18\div15}=0000_2$*, pe ieşirea multiplexorului din figura 3.38, rezultă* $f=\overline{MIR}_{12}$

-În aceste condiţii, la ieşirea porţii SAU-EXCLUSIV (din figura 3.38) rezultă:

$$g=\overline{MIR}_{12}\oplus MIR_{12}=1$$

Prin acest artificiu, primii succesori din tabelele 3.1 şi respectiv 3.2 devin necondiţionaţi.

3.7.4 Memoria de Microprogram şi registrele MAR şi MIR

Memoria de microprogram (MPM) va fi o memorie de tip ROM integrată în procesor şi în care se va „arde" (se va stoca) microprogramul de emulare al setului de instrucţiuni aferent maşini virtuale emulate. În cazul nostru maşina emulată este procesorul didactic iar setul de instrucţiuni emulat va fi setul procesorului didactic stabilit în capitolul 2 (tabelele 2.2 şi 2.3). La 3.4 am adoptat trei tipuri de microinstrucţiuni (de ramificaţie, de transfer sincron şi de transfer asincron) şi am stabilit formatul acestor microinstrucţiuni. Pe baza codificării adoptate, au rezultat microinstrucţiuni cu lungimea de 20 biţi. Prin urmare locaţiile MPM vor fi pe 20 biţi. Capacitatea MPM (numărul de locaţii necesare) va rezulta după proiectarea microprogramului de emulare (paragraful 3.8.2). Prin urmare, MPM va fi un plan de memorie ROM adresat de către registrul MAR (figura 3.31). În starea ST_0 a automatului SEQ, microinstrucţiunea selectată în MPM de microadresa prezentă în MAR, este extrasă (citită) şi încărcată în MIR (*fetch*–ul microinstrucţiunii). Pentru performanţă timpul de acces la MPM trebuie să fie cât mai mic; timpul de acces reprezintă o componentă esenţială care determină perioada semnalului de tact aferent automatului SEQ (aferent procesorului). Cu cât timpul de acces va fi mai mic, cu atât perioada tactului va fi mai mică, respectiv frecvenţa tactului mai mare.

Registrul MAR adresează locaţiile MPM (microinstrucţiunile). Acest registru trebuie să execute două operaţii:

- încărcare paralelă (la activarea comenii LdMAR); această operaţie este necesară pentru execuţia succesorilor JUMP şi respectiv JUMPI.

- incrementare cu 1 (la activarea comenzii +1MAR); această operaţie este necesară pentru execuţia succesorului STEP.

Prin urmare, registrul MAR va fi un registru capabil să încarce paralel şi să numere în sus (*count-up*).

Registrul MIR preia microinstrucţiunile extrase din MPM (*fetch*-ul microinstrucţiunii în starea ST_0 a automatului SEQ). Prin urmare, MIR va trebui să execute o singură operaţie: încărcare paralelă (la activarea comenii LdMIR).

<u>Observaţie</u>:
Este foarte important să se facă distincţie între cele 2 nivele: nivelul instrucţiunii şi cel al microinstrucţiunii. Unitatea de control microprogramată (BGC –ul microprogramat) are o

structură de procesor primar iar instrucţiunile cu care operează acest procesor se numesc microinstrucţiuni. Microinstrucţiunile sunt organizate în microrutine; tabelele de comenzi de la 2.5, 2.6. 2.7 şi 2.8 (de la unitatea de control cablată) se transpun în microrutine (la unitatea microprogramată).

Microinstrucţiunea codifică în câmpurile ei microcomenzi. Fiecare microinstrucţiune generează un set de comenzi C_L *care, în accepţiunea microprogramării, se numesc microcomenzi. Acestui set de microcomenzi îi corespunde, în cazul cablat, setul de comenzi consemnat pe o anumită coloană a tabelului de comenzi echivalent (pe un anumit impuls de stare).*

Procesorul secundar, situat pe un nivelul ierarhic superior celui primar, devine o maşină virtuală emulată (în cazul nostru, maşina virtuală este procesorul didactic). Setul de instrucţiuni aferent maşinii virtuale va fi emulat prin microrutine (secvenţe de microinstrucţiuni). Va exista o microrutină fetch instrucţiune, microrutine fetch operand (câte una pentru fiecare mod de adresare) şi microrutine de execuţie (câte una pentru fiecare instrucţiune din setul de instrucţiuni aferent maşinii virtuale emulate). Procesarea unei instrucţiuni în maşina virtuală constă, în fapt, în parcurgerea unui lanţ de microrutine: microrutina fetch instrucţiune, microrutina fetch operand şi microrutina de execuţie aferentă instrucţiunii respective. Este evident faptul că microrutina fetch instrucţiune este corespondenta fazei IF (din varianta cablată), microrutina fetch operand este corespondenta fazei OF iar microrutina de execuţie este corespundenta fazei EX.

Avantajul esenţial al microprogramării este flexibilitatea: instruciunile maşinii virtuale pot fi modificate prin simpla modificare a microrutinelor de execuţie. Pot fi adăugate instrucţiuni noi prin simpla adăugare în MPM a microrutinelor de execuţie aferente acestor instrucţiuni. În fapt, chiar conţinutul MPM (microprogramul „ars" în MPM) defineşte maşina virtuală emulată; modificând microrutinele poate fi modificată (eventual chiar radical) maşina virtuală emulată. Utilizatorul (inclusiv cel mai avizat utilizator, de la nivelul limbajului de asmblare), "vede" doar maşina virtuală emulată, nu şi unitatea de control microprogramată (procesorul primar) care o emulează.

Redăm în tabelul următor câteva dintre elementele echivalente, plasate pe cele 2 nivele:

Nivelul microinstrucţiunii	***Echivalent la nivelul instrucţiunii***
Memoria de Microprogram (MPM)	*Memoria Principală (MP)*
Microaddress Register (MAR)	*Registru pointer de instrucţiune (PC)*
Microinstruction Register (MIR)	*Registrul Instrucţiunii (IR)*

Procesoarele RISC au de regulă unităţi de control (BGC-uri) cablate. Setul redus de instrucţiuni (uniforme şi simple) conduce la un BGC de complexitate redusă, care poate fi implementat eficient în variantă cablată. Procesoarele CISC au set amplu, complex şi extrem de divers de instrucţiuni. În acest caz se recurge de regulă la microprogramare.

3.8 Microprogramul de emulare

Microprogramul de emulare emulează setul de instrucţiuni aferent procesorului virtual (în cazul nostru setul de instrucţiuni aferent procesorului didactic, stabilit la 2.2). Microprogramul este stocat (este „ars") în MPM şi este format dintr-o colecţie de microrutine.

3.8.1 Organigrama de principiu a microprogramului de emulare

Organigrama microprogramului de emulare este redată în figura 3.39.

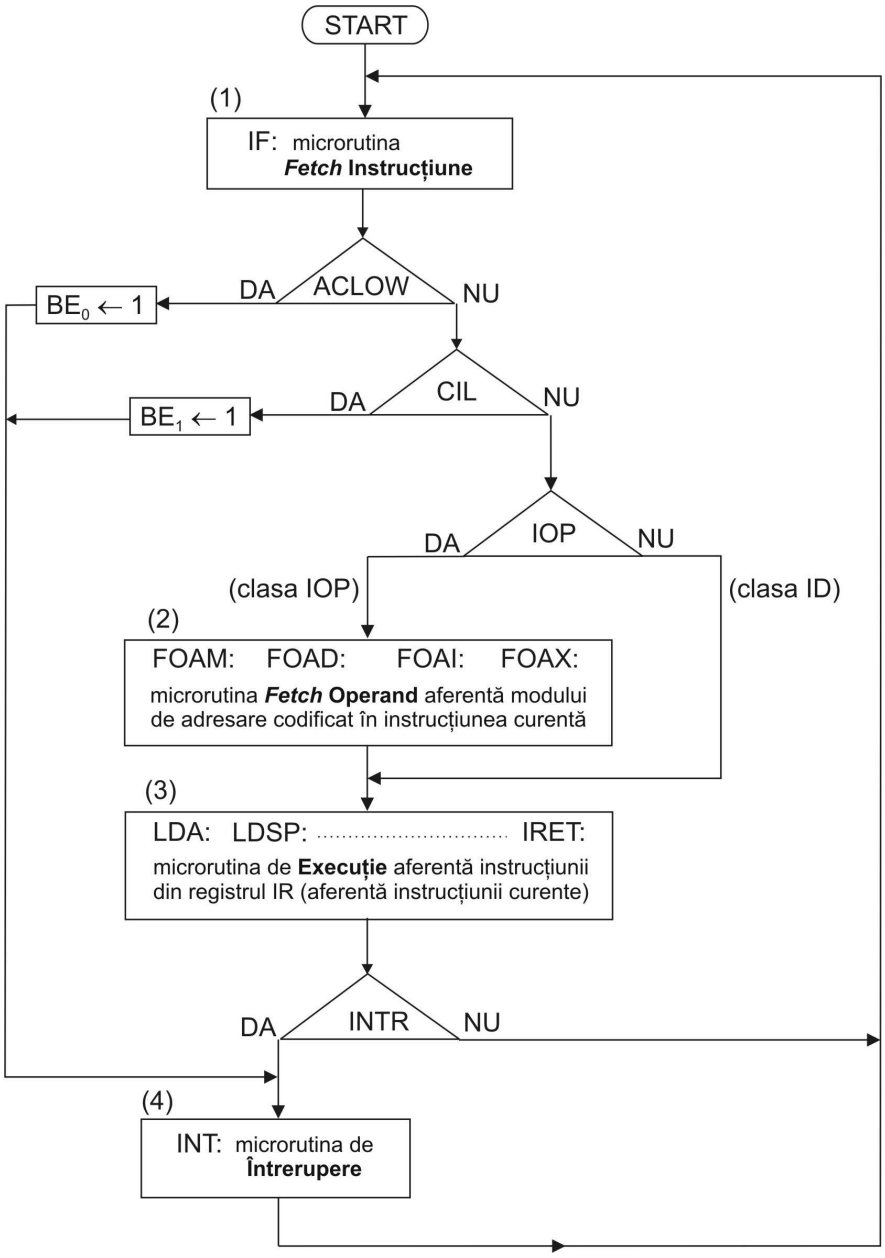


Fig 3.39 Organigrama microprogramului de emulare

Emularea unei instrucţiuni începe întotdeauna cu microrutina ***Fetch* Instrucţiune** (blocul (1) din figura 3.39). Microrutina ***Fetch* Instrucţiune** începe la eticheta IF şi este, de altfel, prima parcursă şi la iniţializarea sistemului (ca urmare a punerii sub tensiune sau ca urmare a acţionării butonului RESET). La finele microrutinei ***Fetch* Instrucţiune** se testează două condiţii de excepţie: ACLOW (căderea tensiunii alternative din reţeaua de alimentare) şi respectiv CIL (cod ilegal). În cazul în care se detectează o condiţie de excepţie activă (ACLOW=1 sau CIL=1) se setează bistabilul de excepţie asociat şi se trece la microrutina de **Întrerupere** (4), care începe la eticheta INT. Setarea bistabilului de excepţie are drept scop generarea vectorului de excepţie asociat excepţiei activate iar microrutina de **Întrerupere** va suspenda temporar programul în curs şi va încărca în registrul PC adresa de start aferentă altui program (aferentă *handler*-ului de excepţie asociat excepţiei activate). Din microrutina de **Întrerupere** se revine la microrutina ***Fetch* Instrucţiune** pentru *fetch*-ul primei instrucţiuni din *handler*.

Dacă la finele microrutinei ***Fetch* Instrucţiune** nu se detectează nici o excepţie activă, se testează condiţia IOP. Dacă IOP=1 (dacă microrutina ***Fetch* Instrucţiune**, anterior parcursă, a încărcat în IR o instrucţiune din clasa IOP), se trece la microrutina ***Fetch* Operand** (2) aferentă modului de adresare codificat în instrucţiunea din IR. Dacă IOP=0 (dacă microrutina ***Fetch* Instrucţiune**, anterior parcursă, a încărcat în IR o instrucţiune din clasa ID), se trece la microrutina de **Execuţie** (3) aferentă instrucţiunii din IR. Prin urmare, pentru emularea unei instrucţiuni din clasa IOP se parcurge un lanţ format din trei microrutine (***Fetch* Instrucţiune**, ***Fetch* Operand** şi **Execuţie**) iar pentru emularea unei instrucţiuni din clasa ID se parcurg doar două microrutine (***Fetch* Instrucţiune** şi **Execuţie**).

La finele microrutinei de **Execuţie** se implementează punctul interuptibil (se testează cererea globală de întrerupere INTR). Dacă INTR=0 (dacă nici un periferic nu solicită întrerupere), se revine la microrutina ***Fetch* Instrucţiune** (pentru *fetch*-ul următoarei instrucţiuni din cadrul programului aflat în execuţie). Dacă INTR=1 (dacă cel puţin un periferic solicită întrerupere), se trece la microrutina de **Întrerupere** (4) prin care se va suspenda temporar programul în curs şi se va încărca în PC adresa primei instrucţiuni din cadrul altui program (din cadrul *handler*-ului de întrerupere asociat perifericului care a întrerupt). De la finele microrutinei de **Întrerupere** se va reveni (necondiţionat) la microrutina ***Fetch* Instrucţiune** (unde se va face *fetch*-ul primei instrucţiuni din cadrul *handler*-ului de întrerupere).

3.8.2 Conţinutul microprogramului de emulare

Microprogramul de emulare aferent setului de instrucţiuni (tabelul 3.16) reprezintă microcodul stocat („ars”) în memoria de microprogram. Acesta emulează setul de instrucţiuni aferent procesorului didactic.

În cadrul acestui microprogram pot fi identificate microinstrucţiuni de ramificaţie, de transfer sincron şi de transfer asincron (cele trei tipuri de microinstrucţiuni definite la 3.4). Microinstrucţiunile sunt specificate într-un limbaj mnemonic, similar limbajului de asamblare. Limbajul mnemonic de specificare (pentru cele trei tipuri de microinstrucţiuni) a fost descris la finalul paragrafelor 3.4.1, 3.4.2.A şi 3.4.2.B. Vom prezenta în continuare doar câteva exemple edificatoare:

- **Microinstrucţiuni de ramificaţie**:

JUMP INT	;salt necondiţionat la eticheta (microadresa) ;INT.
IF NINTR JUMP IF ELSE STEP	;dacă NINTR=1 (dacă INTR=0) salt la eticheta ;(microadresa) IF, dacă nu STEP.
JUMPI LDA+(IR_{14}÷IR_{10},0,0)	;salt indexat la eticheta (microadresa) LDA ;la care (vezi 3.4.1) se adună ;$INDEX_2$=0 0 0 0 0 IR_{14} IR_{13} IR_{12} IR_{11} IR_{10} 0 0_2.
IF IOP JUMPI FOAM+(IR_9,IR_8,0,0) ELSE STEP	;dacă IOP=1 salt indexat la eticheta ;(microadresa) FOAM la care se adună ;$INDEX_3$=0 0 0 0 0 0 0 0 IR_9 IR_8 0 0_2, ; dacă nu STEP (vezi 3.4.1).

- **Microinstrucţiuni de transfer sincron**:

A AND M, PdALU, PmA	;realizează operaţia A←A•M (produs logic pe ;16 biţi).
A SUM M, PdALU, PmA, PdCOND	;realizează adunarea A←A+M şi setarea ;*flag*-urilor de condiţii din registrul FLAG ;în acord cu rezultatul adunării.
A SUM $\overline{M}$, PdALU, PmA, (C_{in},PdCOND)	;realizează scăderea A←A–M ;(scădere prin adunarea la descăzut a ;complementului faţă de 2 al scăzătorului) ;şi poziţionarea *flag*-urilor de condiţii din ;registrul FLAG în acord cu rezultatul ;scăderii.
A SUM NONE, PdALU, PmA, (C_{in},PdCOND)	;realizează adunarea A←A+1 (A←A+C_{in}) şi ;poziţionarea *flag*-urilor de condiţii din ;registrul FLAG în acord cu rezultatul ;adunării.
+2SP	;realizează operaţia de incrementare cu 2 ;a pointerului de stivă (SP←SP+2)
(INTRACK,–2SP)	;realizează achitarea întreruperii şi operaţia ;de decrementare cu 2 a pointerului de stivă ;(SP←SP–2)

- **Microinstrucţiuni de transfer asincron**:

IFCH, +2PC	;realizează *fetch*-ul instrucţiunii din memorie ;(IR←MEM|ADR) şi incrementarea ;registrului PC cu 2 (PC←PC+2).
READ	;realizează citirea operandului din memorie ;şi încărcarea acestuia în registrul M ;(M←MEM|ADR).
A SUM M, PdALU, WRITE	;scrie valoarea A+M în locaţia selectată ;de adresa existentă în registrul ADR ;(MEM|ADR←A+M).
DDA	;relizează deplasarea aritmetică la dreapta ;cu o poziţie a registrului A.
A(1)BVI	;realizează setarea *flag*-ului BVI ;(validarea întreruperilor).

Pe baza exemplelor de mai sus putem concluziona regulile limbajului mnemonic adoptat:

- mnemonica microinstrucţiunii conţine, în ordine, toate microcomenzile codificate în microinstrucţiunea respectivă, utilizând ca separator virgula.

- la microinstrucţiunile de transfer am adoptat o excepţie sugestivă: comenzile codificate în câmpurile SURSA SBUS, OPERAŢIE ALU şi SURSA DBUS sunt specificate printr-o expresie. Expresia sugerează operandul 1 (prelual de pe SBUS), operandul 2 (preluat de pe DBUS) şi operaţia ALU (efectuată asupra celor doi operanzi).

  Exemple:

  1. Expresia A SUM M din microinstrucţiunea de la microadresa 028_H semnifică:

     - microcomanda PdA codificată în câmpul SURSA SBUS
     - microcomanda PdM codificată în câmpul SURSA DBUS
     - microcomanda SUM codificată în câmpul OPERAŢIE ALU

     Cele trei microcomenzi determină efectuarea operaţiei de adunare A+M în ALU.

  2. Expresia NONE DBUS PC din microinstrucţiunea de la microadresa 000_H semnifică:

     - microcomanda NONE codificată în câmpul SURSA SBUS
     - microcomanda PdPC codificată în câmpul SURSA DBUS

- microcomanda DBUS codificată în câmpul OPERAŢIE ALU

Cele trei microcomenzi determină trecerea nemodificată prin ALU a continutului registrului PC.

3. Expresia IR[IND] SUM RG din microinstrucţiunea de la microadresa 016_H semnifică:

   - microcomanda PdIR[IND] codificată în câmpul SURSA SBUS
   - microcomanda PdRG codificată în câmpul SURSA DBUS
   - microcomanda SUM codificată în câmpul OPERAŢIE ALU

   Cele trei microcomenzi determină efectuarea în ALU operaţiei de adunare:

   $$IR[IND]+RG|_{\text{selectat de câmpul ADR.REG. din codul instrucţiunii curente}}.$$

   În cazul modului de adresare indexat, prin această operaţie de adunare se obţine adresa locaţiei de memorie care găzduieşte operandul instrucţiunii (vezi figura 2.5.d). cu explicaţiile aferente).

4. Expresia A SUM $\overline{M}$ din microinstrucţiunea de la microadresa $02A_H$ semnifică:

   - microcomanda PdA codificată în câmpul SURSA SBUS
   - microcomanda Pd$\overline{M}$ codificată în câmpul SURSA DBUS
   - microcomanda SUM codificată în câmpul OPERAŢIE ALU

   Împreună cu microcomanda C_{in} (codificată în câmpul OTHER OPERATIONS), cele trei microcomenzi determină efectuarea operaţia de scădere A-M în ALU.

5. Expresia A XOR M din microinstrucţiunea de la microadresa 036_H semnifică:

   - microcomanda PdA codificată în câmpul SURSA SBUS
   - microcomanda PdM codificată în câmpul SURSA DBUS
   - microcomanda XOR codificată în câmpul OPERAŢIE ALU

   Cele trei microcomenzi determină efectuarea operaţiei logice SAU EXCLUSIV în ALU (operaţia $A \oplus M$).

Microprogramul de emulare este prezentat sub formă tabelară (tabelul3.16). La fiecare microinstrucţiune, pe coloana de comentarii sunt redate explicaţii relative la operaţia efectuată (operaţiile efectuate) de respectiva microinstrucţiune.

Microprogramul de emulare (tabelul 3.16)

ETICHETA	μADRESA (hexa)	CONŢINUT MPM (mnemonica microinstrucţiunii)	COMENTARII
IF:	000_H	NONE DBUS PC, PdALU, PmADR	; ADR←PC (încărcarea adresei de *fetch* în registrul ADR)
	001_H	IFCH, +2PC	; IR←MEM|$_{ADR}$ (*fetch* instrucţiune) şi PC←PC+2 (incrementare PC)
	002_H	IF ACLOW JUMP PWFAIL else STEP	; test ACLOW (punct de ramificaţie în 2 direcţii)
	003_H	IF CIL JUMP ILLEGAL ELSE STEP	; test CIL (punct de ramificaţie în 2 direcţii)
	004_H	IF IOP JUMPI FOAM+(IR_9,IR_8,0,0) else STEP	; dacă IOP=1: ramificaţie în 4 direcţii (spre μrutinele *fetch* operand) ; dacă IOP=0: STEP
	005_H	JUMPI INCA+(IR_{14}÷IR_{10},0,0)	; punct de ramificaţie în 32 direcţii (spre μrutinele de execuţie aferente instrucţiunilor ; din clasa I0)
PWFAIL:	006_H	A(1) BE_0	; BE_0←1 (pentru generarea vectorului de excepţie aferent)
	007_H	JUMP INT	; salt la microrutina de întrerupere (salt la eticheta INT)
ILLEGAL:	008_H	A(1)BE_1	; BE_1←1 (pentru generarea vectorului de excepţie aferent)
	009_H	JUMP INT	; salt la microrutina de întrerupere (salt la eticheta INT)
FOAM:	$00A_H$	IR[OP] SBUS NONE, PdALU, PmM	; M←IR[OP] (operandul imediat)
	$00B_H$	JUMPI LDA+(IR_{14}÷IR_{10},0,0)	; punct de ramificaţie în 32 direcţii (spre μrutinele de execuţie aferente instrucţiunilor ; din clasa IOP)
	$00C_H$	NONE	
	$00D_H$	NONE	
FOAD:	$00E_H$	NONE DBUS RG, PdALU, PmM	;M←RG|$_{\text{selectat de câmpul ADR.REG. din codul instrucţiunii curente}}$ (M←operandul)
	$00F_H$	JUMPI LDA+(IR_{14}÷IR_{10},0,0)	; punct de ramificaţie în 32 direcţii (spre μrutinele de execuţie aferente instrucţiunilor ; din clasa IOP)
	010_H	NONE	
	011_H	NONE	
FOAI:	012_H	NONE DBUS RG, PdALU, PmADR	;ADR←RG|$_{\text{selectat de câmpul ADR.REG. din codul instrucţiunii curente}}$ (ADR←adresa operandului)
	013_H	READ	; M ← MEM|$_{ADR}$ (M←operandul)

	014_H	JUMPI LDA+(IR_{14}÷IR_{10},0,0)	; punct de ramificaţie în 32 direcţii (spre µrutinele de execuţie aferente instrucţiunilor ; din clasa IOP)	
	015_H	NONE		
FOAX:	016_H	IR[IND] SUM RG, PdALU, PmADR	;ADR←IR[IND]+RG|$_{\text{selectat de câmpul ADR.REG. din codul instrucţiunii curente}}$ (ADR←adresa operandului)	
	017_H	READ	; M ← MEM|$_{ADR}$ (M←operandul)	
	018_H	JUMPI LDA+(IR_{14}÷IR_{10},0,0)	; punct de ramificaţie în 32 direcţii (spre µrutinele de execuţie aferente instrucţiunilor ; din clasa IOP)	
	019_H	NONE		
LDA:	$01A_H$	NONE DBUS M, PdALU, PmA	; A←M (A←operandul)	
INTRTEST:	$01B_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$01C_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$01D_H$	NONE		
LDSP:	$01E_H$	NONE DBUS M, PdALU, PmSP	; SP←M (SP←operandul)	
	$01F_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	020_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	021_H	NONE		
LDFLAG:	022_H	NONE DBUS M, PdALU, PmFLAG	; FLAG←M (FLAG←operandul)	
	023_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	024_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	025_H	NONE		
ADD:	026_H	A SUM M, PdALU, PmA, PdCOND	; A←A+M şi (C/Z/S/D)←(C_{OUT}/Z_R/S_R/DCR) -setare *flag*-uri	
	027_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	028_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	029_H	NONE		
SUB:	$02A_H$	A SUM $\overline{M}$, PdALU, PmA, (C_{in}, PdCOND)	; A←A-M şi (C/Z/S/D)←(C_{OUT}/Z_R/S_R/DCR) -setare *flag*-uri	
	$02B_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$02C_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$02D_H$	NONE		

CMP:	$02E_H$	A SUM $\overline{M}$, NONE, NONE, (C_{in}, PdCOND)	; operaţia de scădere A-M în ALU doar ptentru setarea *flag*-urilor (C/Z/S/D)←(C_{OUT}/Z_R/S_R/DCR)	
	$02F_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	030_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	031_H	NONE		
AND:	032_H	A AND M, PdALU, PmA, PdCOND	; A←A·M şi setare *flag*-uri de condiţii	
	033_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	034_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	035_H	NONE		
XOR:	036_H	A XOR M, PdALU, PmA, PdCOND	; A← $A \oplus M$ şi setare *flag*-uri de condiţii	
	037_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	038_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	039_H	NONE		
OR:	$03A_H$	A OR M, PdALU, PmA, PdCOND	; A← $A \vee M$ şi setare *flag*-uri de condiţii	
	$03B_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$03C_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$03D_H$	NONE		
INC:	$03E_H$	IF AD JUMP INCAD ELSE STEP	; salt la eticheta INCAD pentru INC *src* cu adresare directă	
	$03F_H$	0_s SUM M, PdALU, WRITE, (C_{in}, PdCOND)	; $MEM\|_{ADR}$←M+1 (memorie←operand incrementat) şi setare *flag*-uri	
	040_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	041_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
DEC:	042_H	IF AD JUMP DECAD ELSE STEP	; salt la eticheta DECAD pentru DEC *src* cu adresare directă	
	043_H	-1_s SUM M, PdALU, WRITE, PdCOND	$MEM\|_{ADR}$←M-1 (memorie←operand decrementat) şi setare *flag*-uri	
	044_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	045_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
PUSH:	046_H	-2SP	; SP←SP-2	salvare Registru General în stivă
	047_H	SP SBUS NONE, PdALU, PmADR	; ADR←SP	
	048_H	NONE DBUS RG, PdALU, WRITE	; $MEM\|_{ADR}$←RG	

	049_H	JUMP INTRTEST	; salt la punctul interuptibil (salt la eticheta INTRTEST)	
POP:	$04A_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	restaurare Registru General din stivă
	$04B_H$	READ	; M ← $MEM\|_{ADR}$	
	$04C_H$	NONE DBUS M, PdALU, PmRG	; RG←M	
	$04D_H$	JUMP INTRTEST	; salt la punctul interuptibil (salt la eticheta INTRTEST)	
STA:	$04E_H$	IF AD JUMP STAAD ELSE STEP	; salt la eticheta STAAD pentru ST *dest*,A cu adresare directă	
	$04F_H$	A SBUS NONE, PdALU, WRITE	$MEM\|_{ADR}$←A (ST *dest*,A cu adresare indirectă sau indexată)	
	050_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la μrutina *Fetch* Instrucţiune
	051_H	JUMP INT	; interuptibil	dacă INTR=1, salt la μrutina de Întrerupere
STSP:	052_H	IF AD JUMP STSPAD ELSE STEP	; salt la eticheta STSPAD pentru ST *dest*,SP cu adresare directă	
	053_H	SP SBUS NONE, PdALU, WRITE	$MEM\|_{ADR}$←SP (ST *dest*,SP cu adresare indirectă sau indexată)	
	054_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la μrutina *Fetch* Instrucţiune
	055_H	JUMP INT	; interuptibil	dacă INTR=1, salt la μrutina de Întrerupere
STFLAG:	056_H	IF AD JUMP STFLAGAD ELSE STEP	;salt la eticheta STFLAGAD pentru ST *dest*,FLAG cu adresare directă	
	057_H	PdFLAG, WRITE	$MEM\|_{ADR}$←FLAG (ST *dest*,FLAG cu adresare indirectă sau indexată)	
	058_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la μrutina *Fetch* Instrucţiune
	059_H	JUMP INT	; interuptibil	dacă INTR=1, salt la μrutina de Întrerupere
JC:	$05A_H$	IF NC JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece C=0	
	$05B_H$	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JC *adr* cu adresare imediată	
	$05C_H$	PdADR, PmPC	; PC←ADR (pentru emulare JC *adr* cu mod adresare indirect sau indexat)	
	$05D_H$	JUMP INTRTEST	; salt la punctul interuptibil	
JNC:	$05E_H$	IF C JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece C=1	
	$05F_H$	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JNC *adr* cu adresare imediată	
	060_H	PdADR, PmPC	; PC←ADR (pentru emulare JNC *adr* cu mod adresare indirect sau indexat)	
	061_H	JUMP INTRTEST	; salt la punctul interuptibil	
JZ:	062_H	IF NZ JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece Z=0	
	063_H	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JZ *adr* cu adresare imediată	
	064_H	PdADR, PmPC	; PC←ADR (pentru emulare JZ *adr* cu mod adresare indirect sau indexat)	

	065_H	JUMP INTRTEST	; salt la punctul interuptibil	
JNZ:	066_H	IF Z JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece Z=1	
	067_H	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JNZ *adr* cu adresare imediată	
	068_H	PdADR, PmPC	; PC←ADR (pentru emulare JNZ *adr* cu mod adresare indirect sau indexat)	
	069_H	JUMP INTRTEST	; salt la punctul interuptibil	
JP:	$06A_H$	IF S JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece S=1	
	$06B_H$	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JP *adr* cu adresare imediată	
	$06C_H$	PdADR, PmPC	; PC←ADR (pentru emulare JP *adr* cu mod adresare indirect sau indexat)	
	$06D_H$	JUMP INTRTEST	; salt la punctul interuptibil	
JM:	$06E_H$	IF NS JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece S=0	
	$06F_H$	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JM *adr* cu adresare imediată	
	060_H	PdADR, PmPC	; PC←ADR (pentru emulare JM *adr* cu mod adresare indirect sau indexat)	
	071_H	JUMP INTRTEST	; salt la punctul interuptibil	
JV:	072_H	IF NV JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece V=0	
	073_H	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JV *adr* cu adresare imediată	
	074_H	PdADR, PmPC	; PC←ADR (pentru emulare JV *adr* cu mod adresare indirect sau indexat)	
	075_H	JUMP INTRTEST	; salt la punctul interuptibil	
JNV:	076_H	IF V JUMP INTRTEST ELSE STEP	; salt la punctul interuptibil deoarece V=1	
	077_H	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JNV *adr* cu adresare imediată	
	078_H	PdADR, PmPC	; PC←ADR (pentru emulare JNV *adr* cu mod adresare indirect sau indexat)	
	079_H	JUMP INTRTEST	; salt la punctul interuptibil	
JMP:	$07A_H$	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM pentru emulare JMP *adr* cu adresare imediată	
	$07B_H$	PdADR, PmPC	; PC←ADR (PC←adresa de salt); JMP cu adresare indirectă sau indexată	
	$07C_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la μrutina *Fetch* Instrucţiune
	$07D_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la μrutina de Întrerupere
CALL:	$07E_H$	PdADR, PmM, -2SP	; M←ADR; SP←SP-2	(salvare PC în stivă)
	$07F_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	
	080_H	NONE DBUS PC, PdALU, WRITE	; $MEM\|_{ADR}$←PC	
	081_H	JUMP CALL1	; salt la μrutina de execuţie a instrucţiunii JMP *adr*	

INCA:	082_H	A SUM 0_d, PdALU, PmA, (C_{in}, PdCOND)	; A←A+1 şi setare *flag*-uri de condiţii	
	083_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	084_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	085_H	NONE		
DECA:	086_H	A SUM -1_d, PdALU, PmA, PdCOND	; A←A-1 şi setare *flag*-uri de condiţii	
	087_H	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	088_H	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	089_H	NONE		
NEGA:	$08A_H$	A $\overline{SBUS}$ NONE, PdALU, PmA, Pd COND	; A← $\overline{A}$ şi setare *flag*-uri de condiţii	
	$08B_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$08C_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$08D_H$	NONE		
PUSHA:	$08E_H$	-2SP	; SP←SP-2	STIVĂ←A
	$08F_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	
	090_H	A SBUS NONE, PdALU, WRITE	; $MEM\|_{ADR}$←A	
	091_H	JUMP INTRTEST	; salt la punct interuptibil	
PUSHFLAG:	092_H	-2SP	; SP←SP-2	STIVĂ←FLAG
	093_H	SP SBUS NONE, PdALU, PmADR	; ADR←SP	
	094_H	PdFLAG, WRITE	; $MEM\|_{ADR}$←FLAG	
	095_H	JUMP INTRTEST	; salt la punct interuptibil	
POPA:	096_H	SP SBUS NONE, PdALU, PmADR	; ADR←SP	A←STIVĂ
	097_H	READ	; M←$MEM\|_{ADR}$	
	098_H	NONE DBUS M, PdALU, PmA, +2SP	; A←M, SP←SP+2	
	099_H	JUMP INTRTEST	; salt la punctul interuptibil	
POPFLAG:	$09A_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	FLAG←STIVĂ
	$09B_H$	READ	; M←$MEM\|_{ADR}$	
	$09C_H$	NONE DBUS M, PdALU, PmFLAG, +2SP	; FLAG←M, SP←SP+2	
	$09D_H$	JUMP INTRTEST	; salt la punctul interuptibil	

ASL:	$09E_H$	DSA	; deplasare la stânga aritmetică a registrului acumulator	
	$09F_H$	A SBUS NONE, PdALU, $PdCOND_1$	; poziţionare *flag*-uri Z, S şi V (mai puţin C care trebuie să rămână în ; starea în care a fost setat de microinstrucţiunea anterioară!)	
	$0A0_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0A1_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
ASR:	$0A2_H$	DDA	; deplasare la dreapta aritmetică a registrului acumulator	
	$0A3_H$	A SBUS NONE, PdALU, $PdCOND_1$	; poziţionare *flag*-uri Z, S şi V (mai puţin C care trebuie să rămână în ; starea în care a fost setat de microinstrucţiunea anterioară!)	
	$0A4_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0A5_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
LSR:	$0A6_H$	DDL	; deplasare la dreapta logică a registrului acumulator	
	$0A7_H$	A SBUS NONE, PdALU, $PdCOND_1$	; poziţionare *flag*-uri Z, S şi V (mai puţin C care trebuie să rămână în ; starea în care a fost setat de microinstrucţiunea anterioară!)	
	$0A8_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0A9_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
ROL:	$0AA_H$	RS	; rotire la stânga a registrului acumulator (rotire fără *Carry*)	
	$0AB_H$	IF NINTR JUMP IF ELSE STEP	; punct	INTR=0; salt la *Fetch* Instrucţiune
	$0AC_H$	JUMP INT	; interuptibil	INTR=1; salt la µrutina de Întrerupere
	$0AD_H$	NONE		
ROR:	$0AE_H$	RD	; rotire la dreapta a registrului acumulator (rotire fără *Carry*)	
	$0AF_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0B0_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$0B1_H$	NONE		
RLC:	$0B2_H$	RSC	; rotire la stânga a registrului acumulator (rotire cu *Carry*)	
	$0B3_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0B4_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$0B5_H$	NONE		
RRC:	$0B6_H$	RDC	; rotire la dreapta a registrului acumulator (rotire cu *Carry*)	

	$0B7_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0B8_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$0B9_H$	NONE		
NOP:	$0BA_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0BB_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$0BC_H$	NONE		
	$0BD_H$	NONE		
HALT:	$0BE_H$	A(0)BP0	; resetare Bistabil de Pornire-Oprire (Oprirea procesorului prin blocare tact procesor!)	
JMPAM:	$0BF_H$	IR[OP] SUM PC, PdALU, PmPC	; PC←PC+IR[OP] (PC←adresa de salt); salt relativ la adresare imediată	
	$0C0_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0C1_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
EI:	$0C2_H$	A(1)BVI	; BVI←1 (setarea *flag*-ului de validare a întreruperilor)	
	$0C3_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0C4_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
	$0C5_H$	NONE		
DI:	$0C6_H$	A(0)BVI	; BVI←0 (resetarea *flag*-ului de validare a întreruperilor)	
	$0C7_H$	JUMP IF	; salt la µicrorutina *Fetch* Instrucţiune	
	$0C8_H$	NONE		
	$0C9_H$	NONE		
RET:	$0CA_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	; PC←STIVĂ
	$0CB_H$	READ	; M←MEM_{ADR}	
	$0CC_H$	NONE DBUS M, PdALU, PmPC, +2SP	; PC←M, SP←SP+2	
	$0CD_H$	JUMP INTRTEST	; salt la punct interuptibil	
IRET:	$0CE_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	; PC←STIVĂ
	$0CF_H$	READ	; M←MEM_{ADR}	
	$0D0_H$	NONE DBUS M, PdALU, PmPC, +2SP	; PC←M, SP←SP+2	
	$0D1_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	;FLAG←STIVĂ
	$0D2_H$	READ	; M←MEM_{ADR}	

	$0D3_H$	NONE DBUS M, PdALU, PmFLAG, +2SP	; FLAG←M, SP←SP+2	
	$0D4_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0D5_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
INCAD:	$0D6_H$	0_S SUM M, PdALU, PmRG, (C_{in}, PdCOND)	; RG←M+1 şi setare *flag*-uri de condiţii	
	$0D7_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0D8_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
DECAD:	$0D9_H$	-1_S SUM M, PdALU, PmRG, PdCOND	; RG←M-1 şi setare *flag*-uri de condiţii	
	$0DA_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0DB_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
STAAD:	$0DC_H$	A SBUS NONE, PdALU, PmRG	; RG←A	
	$0DD_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0DE_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
STSPAD:	$0DF_H$	SP SBUS NONE, PdALU, PmRG	; RG←SP	
	$0E0_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0E1_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
STFLAGAD:	$0E2_H$	PdFLAG, PmRG	; RG←FLAG	
	$0E3_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la µrutina *Fetch* Instrucţiune
	$0E4_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la µrutina de Întrerupere
INT:	$0E5_H$	(INTA, -2SP)	; SP←SP-2	; INTA -achitare întrerupere ; STIVĂ←FLAG
	$0E6_H$	SP SBUS NONE, PdALU,PmADR	; ADR←SP	
	$0E7_H$	PdFLAG, WRITE	; $MEM\|_{ADR}$←FLAG	
	$0E8_H$	-2SP	; SP←SP-2	; STIVĂ←PC
	$0E9_H$	SP SBUS NONE, PdALU, PmADR	; ADR←SP	
	$0EA_H$	NONE DBUS PC, PdALU, WRITE	; $MEM\|_{ADR}$←PC	
	$0EB_H$	PdVI, PmADR	; ADR←VI (ADR←vector întrerupere/excepţie)	
	$0EC_H$	READ, (A(0)BI, A(0)BE)	; M←$MEM\|_{ADR}$ (M←adresă start *handler*), BI←0, BE←0	
	$0ED_H$	NONE DBUS M, PdALU, PmPC	; PC←M (PC←adresă start *handler* întrerupere/excepţie)	
	$0EE_H$	JUMP IF	; salt la µrutina *Fetch* Instrucţiune (salt la eticheta IF)	

CALL1:	$0EF_H$	IF AM JUMP JMPAM ELSE STEP	; salt la eticheta JMPAM ptentru emulare CALL *adr* cu adresare imediată	
	$0F0_H$	NONE DBUS M, PdALU, PmPC	; PC←M (PC←adresa de CALL); CALL cu adresare indirectă sau indexată	
	$0F1_H$	IF NINTR JUMP IF ELSE STEP	; punct	dacă INTR=0, salt la μrutina *Fetch* Instrucţiune
	$0F2_H$	JUMP INT	; interuptibil	dacă INTR=1, salt la μrutina de Întrerupere
	$0F3_H$	NONE	; spaţiu neutilizat (disponibil pentru extensia setului de instrucţiuni)	
	$0F4_H$	NONE		
	$0F5_H$	NONE		
	$0F6_H$	NONE		
	$0F7_H$	NONE		
	$0F8_H$	NONE		
	$0F9_H$	NONE		
	$0FA_H$	NONE		
	$0FB_H$	NONE		
	$0FC_H$	NONE		
	$0FD_H$	NONE		
	$0FE_H$	NONE		
	$0FF_H$	NONE		

Tabelul 3.16 Microprogramul de emulare

Microprogramul de emulare (tabelul 3.16) conţine o colecţie de microrutine. Legăturile dintre microrutine sunt realizate prin microinstrucţiuni de ramificaţie. Colecţia cuprinde următoarele microrutine:

A. Microrutina *Fetch* Instrucţiune

Microrutina ***Fetch* Instrucţiune** începe la microadresa 000_H (la eticheta IF) şi conţine 4 microinstrucţiuni. Această microrutină controlează următoarele operaţii succesive:

- adresa locaţiei de memorie din care urmează să se extragă codul instrucţiunii, este transferată din registrul PC în registrul ADR;
- se citeşte codul instrucţiunii din memorie, codul citit se încarcă în registrul IR (microcomanda IFCH) şi se pregăteşte (anticipat) în registrul PC adresa următoarei instrucţiuni (microcomanda +2PC); În continuare, instrucţiunea al cărui cod a fost încărcat în registrul IR şi care urmează a fi procesată (emulată în microcod) o vom denumi **instrucţiune curentă**.
- dacă se detectează ACLOW=1, se execută un salt la eticheta PWFAIL, unde se setează bistabilul de excepţie asociat (BE_0) şi de unde se execută un nou salt la microrutina de **Întrerupere**;
- dacă în registrul IR s-a încărcat o instrucţiune ilegală (dacă CIL=1), se execută salt la eticheta ILLEGAL, unde se setează bistabilul de excepţie asociat (BE_1) şi de unde se execută un nou salt la microrutina de **Întrerupere**;

Instrucţiunea curentă poate fi din clasa IOP sau din clasa ID. Prin urmare, din microrutina ***Fetch* Instrucţiune** urmează să se treacă la:

- microrutina ***Fetch* Operand** aferentă modului de adresare codificat în câmpul MA din codul înstrucţiunii curente (biţii IR_9, IR_8), dacă instrucţiunea curentă este din clasa IOP.
- microrutina de **Execuţie** aferentă instrucţiunii curente, dacă instrucţiunea curentă este din clasa ID (instrucţiunile din clasa ID sunt instrucţiuni fără operanzi şi prin urmare, din microrutina ***Fetch* Instrucţiune** se sare direct la microrutina de **Execuţie**).

Legătura spre microrutina următoare este implementată prin microinstrucţiunile de la microadresele 004_H şi respectiv 005_H). Cele două microinstrucţiuni, precum şi mecanismul de legătură (de ramificaţie) deosebit de sofisticat pe care îl implementează, au fost explicate la 3.4.1.B (vezi figura 3.3 şi explicaţiile aferente).

B. Microrutinele *Fetch* Operand

Microrutinele ***Fetch* Operand** încep la etichetele FOAM, FOAD, FOAI şi respectiv FOAX (câte o microrutină pentru fiecare mod de adresare). Cele 4 microrutine localizează

operandul conform procedurilor indicate la 2.2.1 (vezi figura 2.5 şi explicaţiile aferente), îl citesc (din memorie dacă e cazul) şi îl încarcă în registrul M.

Microrutina ***Fetch* Operand** (aferentă modului de adresare codificat în instrucţiunea curentă) este parcursă numai dacă instrucţiunea curentă aparţine clasei IOP; ea urmează imediat după microrutina ***Fetch* Instrucţiune**.

Toate cele 4 microrutine ***Fetch* Operand** se încheie cu microinstrucţiunea de salt indexat JUMPI LDA+(IR_{14}÷IR_{10}, 0, 0) (vezi microadresele $00B_H$, $00F_H$, 014_H şi 018_H din tabelul 3.16). Prin această microinstrucţiune (de salt indexat) se stabileşte legătura spre microrutina de execuţie (se execută un salt direct pe începutul microrutinei de execuţie aferentă instrucţiunii curente).

Prin urmare, la sfârşitul celor 4 microrutine *fetch* operand sunt implementate 4 puncte de ramificaţie identice (vezi figura 3.3 şi explicaţiile aferente). Sunt puncte de ramificaţie în 32 de direcţii (practic doar 26 utilizate), prin care se stabilesc legăturile la microrutinele de execuţie aferente celor 26 instrucţiuni din clasa IOP (tabelul 2.2).

Observaţie:
Instrucţiunile ST dest,A / ST dest,SP / ST dest,FLAG nu citesc ci scriu operandul în memorie (aceste instrucţiuni specifică un operand destinaţie şi nu unul sursă). Pentru modurile de adresare AI şi AX, adresa locaţiei de memorie ce urmează a fi scrisă (adresa operandului destinaţie) se calculează prin aceleaşi proceduri indicate la 2.2.1 (figurile 2.5 c). şi respectiv 2.5.d).). Citirea operandului destinaţie în cadrul microrutinei ***Fetch Operand*** *este inutilă în aceste cazuri, dar calculul adresei operandului (adresei locaţiei de memorie) este corect şi necesar. La această adresă se va scrie, în microrutina de* ***Execuţie*** *a instrucţiunii, operandul (conţinutul registrului A în microrutina de* ***Execuţie*** *aferentă instrucţiunii ST dest,A, conţinutul registrului SP în microrutina de* ***Execuţie*** *aferentă instrucţiunii ST dest,SP şi respectiv conţinutul registrului FLAG în microrutina de* ***Execuţie*** *aferentă instrucţiunii ST dest,FLAG). Pentru uniformitate şi simplificare, se vor utiliza aceleaşi microrutine* ***Fetch Operand*** *şi la instrucţiunile ST dest,A / ST dest,SP / ST dest,FLAG, cu observaţia că ultima microinstrucţiune din cadrul acestor microrutine (microinstrucţiunea READ), care citeşte un pretins operand din memorie, este nenecesară dar nu deranjază (decât sub aspectul timpului pierdut pentru execuţia unei microinstrucţiuni inutile). Evident că, pentru cele trei instrucţiuni în discuţie, se pot proiecta microrutine speciale de* ***Calcul Adresă Operand*** *şi nu de* ***Fetch Operand****, introducând (în microprogram) şi mecanismul de ramificaţie necesar. Nu s-a adoptat această soluţie pentru a nu complica microprogramul de emulare (şi în primul rând, pentru că nu ar conduce la îmbunătăţirea performanţelor procesorului). Ultima afirmaţie nu poate fi validată decât printr-o comparare a performanţelor procesorului, în cele două variante de implementare a microcodului. Lăsăm acest exerciţiu interesant pe seama cititorului.*

C. Microrutinele de Execuţie

Microprogramul de emulare (tabelul 3.16) conţine două grupe de microrutine de **Execuţie**: microrutinele de **Execuţie** aferente instrucţiunilor din clasa IOP (instrucţiunile din tabelul 2.2) şi respectiv microrutinele de **Execuţie** aferente instrucţiunilor din clasa ID (instrucţiunile din tabelul 2.3). În prima grupă sunt 26 de microrutine (aferente celor 26 instrucţiuni din clasa IOP) iar în a doua grupă sunt 20 microrutine (aferente celor 20 instrucţiuni din clasa ID). Microrutinele de **Execuţie** din prima grupă urmează (în cadrul microprogramului de emulare) după microrutinele ***Fetch* Operand** (figura 3.3).

Microrutinele de **Execuţie** din a doua grupă urmează (în cadrul microprogramului de emulare) imediat după microrutina ***Fetch* Instrucţiune** (figura 3.3). Legătura între microrutinele ***Fetch* Operand** şi cele 26 microrutinele de **Execuţie** din prima grupă este implementată prin microinstrucţiunea JUMPI LDA+(IR_{14}÷IR_{10},0,0), pe care o regăsim la sfârşitul fiecăreia dintre cele patru microrutine ***Fetch* Operand** (la microadresele $00B_H$, $00F_H$, 014_H şi respectiv 018_H). Legătura între microrutina ***Fetch* Instrucţiune** şi cele 20 microrutine de **Execuţie** din cea de a doua grupă este implementată prin microinstrucţiunea JUMPI INCA+(IR_{14}÷IR_{10},0,0), pe care o regăsim la sfârşitul microrutinei ***Fetch* Instrucţiune** (la microadresa 005_H).

În microrutina de Execuţie aferentă unei anumite instrucţiuni, se execută operaţia solicitată (operaţiile solicitate) de instrucţiunea respectivă. Enumerăm câteva exemple:

- încărcarea operandului *src* în registrul A la instrucţiunea LD A,*src*; această microrutină începe la eticheta LDA (microadresa $01A_H$).

- adunarea operandului *src* la conţinutul registrului A şi poziţionarea *flag*-urilor de condiţii în acord cu rezultatul adunării, la instrucţiunea ADD A,*src*; această microrutină începe la eticheta ADD (microadresa 026_H).

- incrementarea cu 1 a registrului A şi poziţionarea *flag*-urilor de condiţii în acord cu rezultatul incrementării, la instrucţiunna INC A; această microrutină începe la eticheta INCA (microadresa 082_H).

- calculul adresei de salt şi încărcarea acestei adrese în registrul PC, la instrucţiunea JMP *adr*; această microrutină începe la eticheta JMP (microadresa $07A_H$).

Prin urmare, fiecare instrucţiune are propria sa microrutină de execuţie care începe la o etichetă preluată din (sugerată de) mnemonica instrucţiunii respective.

Fiecare microrutină de execuţie se încheie cu punctul interuptibil (exceptând microrutinele aferente instrucţiunilor DI şi respectiv HALT, unde punctul interuptibil nu mai are sens). Punctul interuptibil este implementat prin două microinstrucţiuni de ramificaţie (figura 3.40):

În concluzie, prin punctul interuptibil implementat la finele microrutinelor de execuţie, se face legătura la microrutina următoare, care va fi:

- Microrutina ***Fetch* Instrucţiune**, dacă nici un periferic nu solicită întrerupere (dacă INTR=0). În acest caz, se trece la *fetch*-ul următoarei instrucţiuni din cadrul programului procesat.

- Microrutina de **Întrerupere**, dacă cel puţin un periferic solicită întrerupere (dacă INTR=1). Microrutina de **Întrerupere** va suspenda programul în curs şi va încărca în registrul PC adresa de start a *handler*-ului de întrerupere, aferent perifericului care solicită întrerupere. Fiecare periferic are propriul *handler* iar *handler*-ele de întrerupere sunt programe din cadrul sistemului de operare.

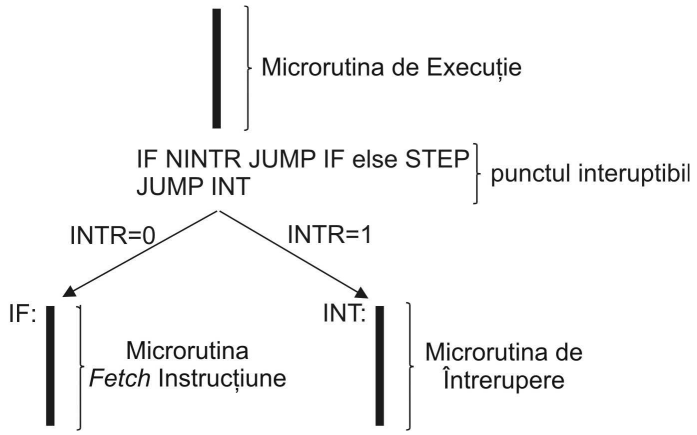


Fig 3.40 Ramificaţia introdusă în microprogramul de emulare prin punctul interuptibil

D. Microrutina de Întrerupere

După cum am arătat mai sus, microrutina de **Întrerupere** este executată dacă, în punctul interuptibil, se detectează INTR=1. În acest caz, microrutina de **Întrerupere** este executată datorită întreruperii activate de un periferic.

Aceeaşi microrutină de **Întrerupere** este executată şi în cazul activării unei excepţii. La procesorul didactic am prevăzut două condiţii de excepţie: ACLOW şi CIL (care sunt testate la sfârşitul microrutinei ***Fetch* Instrucţiune**). La activarea unei excepţii, anterior saltului la microrutina de **Întrerupere** se va seta bistabilul de excepţie asociat respectivei excepţii, asigurându-se prin aceasta generarea vectorului de excepţie asociat excepţiei respective. Să menţionăm că vectorii de întrerupere şi respectiv de excepţie reprezintă entităţi echivalente (pointeri în tabela IVT). Prin urmare, vectorii adresează locaţiile tabelei IVT *(Interrupt Vector Table)* iar în aceste locaţii sunt stocate adresele de start aferente *handler*-elor de întrerupere, respectiv excepţie. În realitate, excepţia este tot o întrerupere (o întrerupere internă). Excepţia poate fi denumită şi întrerupere internă deoarece cauza ei este detectată intern (la nivelul procesorului), spre deosebire de întreruperi care provin din exterior (de la periferice). Să mai notăm că, procesoarele comerciale moderne tratează şi alte condiţii de excepţie, cum ar fi:

- DIV ERR (depăşire de domeniu la împărţire; apare la instrucţiunea de împărţire, dacă împărţitorul este zero)

- SEGMENT FAULT (tentativă de acces în afara segmentului; poate să apară în sistemele care utilizează segmentarea memoriei). Pe această excepţie (ca resursă *hardware*) şi pe *handler*-ul de excepţie aferent (ca resursă *software* din cadrul sistemului de operare) se implementează mecanismul de protecţii (protejarea segmentelor de memorie).

- SEGMENT NOT PRESENT (tentativă de acces într-un segment neprezent în memoria internă; poate să apară în sistemele care utilizează segmentarea memoriei). Pe baza aceastei excepţii şi a *handler*-ului aferent se poate implementa mecanismul memoriei virtuale la nivelul segmentelor (în sistemele de operare care operează numai cu segmentare).

- PAGE FAULT (tentativă de acces într-o pagină neprezentă în memoria internă; poate să apară în sistemele care utilizează paginarea memoriei) Pe această excepţie (ca resursă *hardware*) şi pe *handler*-ul aferent (ca resursă *software* din cadrul sistemului de operare) se implementează mecanismul memoriei virtuale la nivelul paginilor (în sistemele de operare care operează cu paginare).

- STACK FAULT (depăşire stivă; se activează la o depunere în stivă, dacă stiva depăşeşte zona de memorie alocată)

Microrutina de **Întrerupere** începe la microadresa $0E5_H$ (la eticheta INT). În această microrutină se execută următoarele operaţii:

- INTA (achitarea cererii INTR; achitarea va ajunge la *controller*-ul I/O aferent perifericului care a întrerupt. Suplimentar, INTA va încărca vectorul de întrerupere (aferent perifericului care a întrerupt) în registrul IVR)

- STIVĂ←FLAG (salvare în stivă a *flag*-urilor de condiţii din momentul întreruperii)

- STIVĂ←PC (salvare în stivă a adresei de revenire în programul ce va fi suspendat temporar prin întrerupere)

- $PC \leftarrow MEM|_{IVR}$ (încărcare în registrul PC a adresei de start a *handler*-ului de întrerupere)

- JUMP IF (salt la microrutina ***Fetch* Instrucţiune** care urmează să realizeze *fetch*-ul primei instrucţiuni din *handler*). Prin această microinstrucţiune este implementată de fapt legatura (necondiţionată) dintre microrutina de **Întrerupere** şi microrutina următoare (microrutina ***Fetch* Instrucţiune**).

3.9 Rectificări în structura unităţii de control microprogramate

Analizând tabelul 3.16, constatăm că microprogramul de emulare aferent procesorului didactic se întinde între microadresa 000_H (microadresa de start) şi respectiv $0F2_H$ (microadresa ultimei microinstrucţiuni utile). Prin urmare, lungimea microprogramului de emulare este de 243=256-13 microinstrucţiuni. Putem, în sfârşit, să luăm decizia că MPM (memoriea de microprogram) va avea capacitatea de $256=2^8$ locaţii pe 20 biţi (un plan de ROM cu capacitatea 256×20). Prin această decizie ne asigurăm şi o mică rezervă de 13 locaţii (între microadresele $0F3_H$ şi respectiv $0FF_H$) care poate fi utilizată pentru eventuale

rectificări ale microcodului sau pentru eventuale dezvoltări (pot fi eventual adăugate câteva instrucţiuni noi în setul de instrucţiuni aferent procesorului didactic).

Un plan de memorie MPM de $256=2^8$ locaţii revendică 8 biţi de adresă. Consecinţele imediate ce decurg din această decizie sunt:

- Registrul MAR va fi un registru pe 8 biti şi nu pe 12 biţi, cum prevăzusem iniţial. Acest registru îşi va păstra însă caracteristicile funcţionale; va fi un registru capabil să execute două funcţii:

  - funcţia de încărcare paralelă, activată prin comanda LdMAR

  - funcţia de numărare în sus (*count up*), activată prin comanda +1MAR.

- Câmpul MICROADRESA DE SALT codificat în microinstrucţiunea de ramificaţie, va fi pe 8 biţi şi nu pe12 biţi, aşa cum rezervasem iniţial. Această reformulare conduce şi la o uşoară rectificare a formatului microinstrucţiunii de ramificaţie din figura 3.2. Rectificarea este prezentată în figura 3.41.

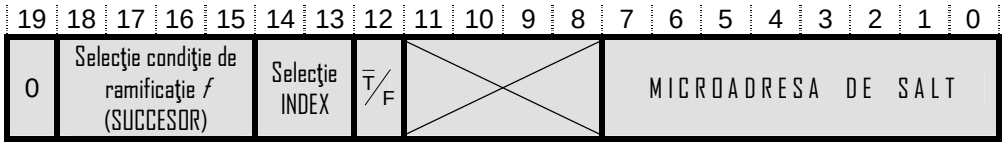


Fig 3.41 Formatul microinstrucţiunii de ramificaţie rectificat

Prin urmare, MICROADRESA DE SALT va fi codificată în câmpul $MIR_{7\div0}$ din formatul microinstrucţiunii de ramificaţie; biţii $MIR_{11\div8}$ vor rămăne neutilizaţi în acest format.

- Sumatorul din componenţa unităţii de control microprogramate (figura 3.31) va fi un sumator pe 8 biţi şi nu pe 12, aşa cum prevăzusem iniţial. Acest sumator va prelua, pe intrarea A, câmpul MICROADRESA DE SALT (pe 8 biţi!) din registrul MIR şi pe intrarea B, INDEX-ul (pe 8 biţi!) generat de BLOCUL DE SELECŢIE INDEX. Rezultatul produs de sumator reprezintă microadresa reală de salt, care va fi încărcată în registrul MAR (la activarea comenzii LdMAR). Evident că această încărcare se produce numai dacă în MIR se află o microinstrucţiune de ramificaţie cu succesor JUMPI sau JUMP (numai în aceste condiţii automatul SEQ activează comanda LdMAR).

- BLOCUL DE SELECŢIE INDEX va genera un INDEX pe 8 biţi şi nu pe 12, aşa cum prevăzusem iniţial. Asta înseamnă că structura acestui bloc, stabilită în figura 3.37, trebuie rectificată uşor. BLOCUL DE SELECŢIE INDEX rectificat este prezentat în figura 3.42.

Sa remarcăm că rectificarea operată la nivelul BLOCULUI DE SELECŢIE INDEX este minoră; practic nu mai sunt emişi biţii $IND_{11 \div 8}$ (care oricum aveau valoarea zero şi în schema din figura 3.37)

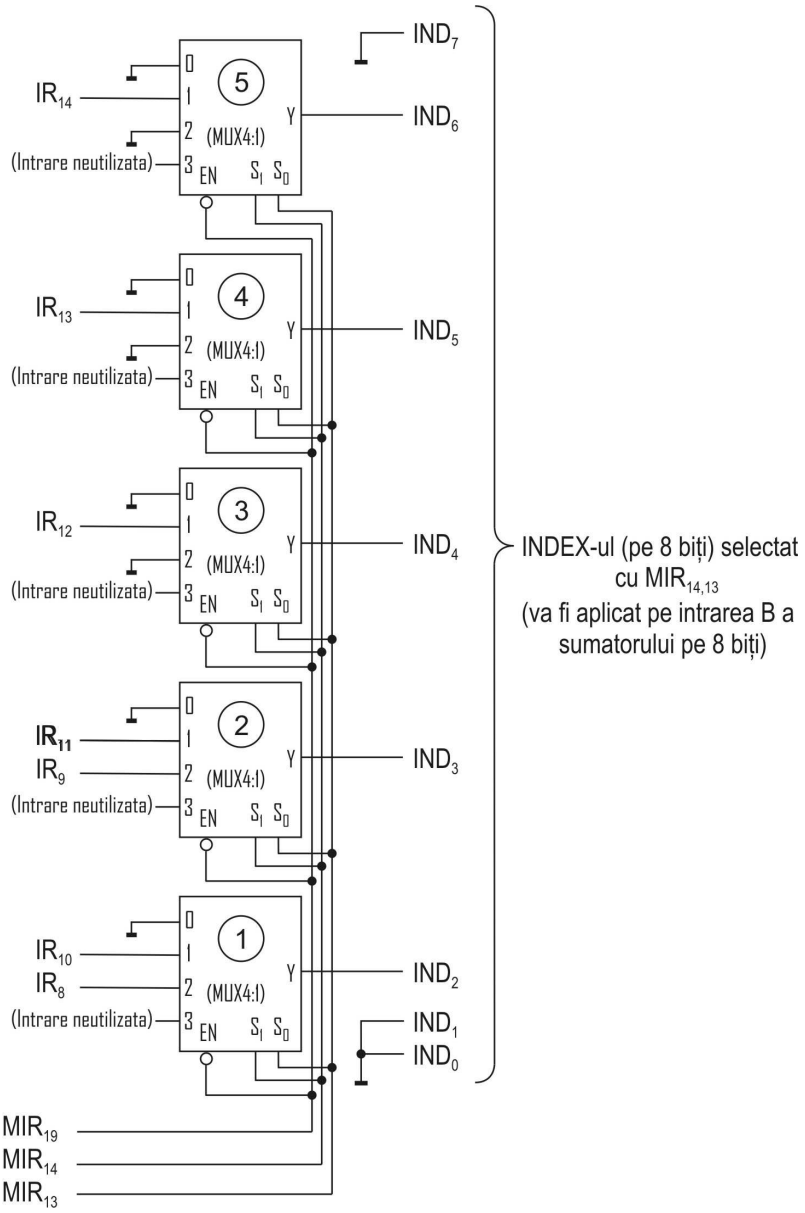


Fig 3.42 Blocul de selecţie index rectificat

3.10 Înlănţuirea microrutinelor în procesul emulării instrucţiunilor

Emularea unei instrucţiuni constă în parcurgerea unui lanţ de microrutine care începe întotdeauna cu microrutina ***Fetch* Instrucţiune** (figura 3.43). Pentru instrucţiunile din clasa IOP, emularea continuă cu microrutina ***Fetch* Operand** (aferentă modului de adresare codificat în codul instrucţiunii respective) şi apoi cu microrutina de **Execuţie** (aferentă instrucţiunii respective). Pentru instrucţiunile din clasa ID, emularea continuă direct cu microrutina de **Execuţie** (aferentă instrucţiunii respective).

În figura 3.43 sunt evidenţiate şi microinstrucţiunile de ramificaţie prin care sunt împlementate legăturile dintre microrutine. Aceste legături, identic implementate, pot fi regăsite şi în microprogramul de emulare din tabelul 3.16.

Reamitim faptul că ciclul instrucţiunii reprezintă timpul în care instrucţiunea este procesată complet. În cazul procesoarelor microprogramate, ciclul instrucţiunii include timpii de execuţie a microrutinelor parcurse pentru emularea instrucţiunii, la care se adaugă timpii de stabilire a legăturilor dintre aceste microrutine. În cazul procesorului didactic, pe baza notaţiilor din figura 3.43, ciclul instrucţiunii (t_C) este dat de ecuaţia (3.30), pentru instrucţiunile din clasa IOP şi respectiv (3.31), pentru instrucţiunile din clasa ID

$$t_C=t_{FI}+t_{leg1}+t_{FO}+t_{leg2}+t_{EX} \qquad (3.30)$$

$$t_C=t_{FI}+t_{leg3}+t_{EX} \qquad (3.31)$$

În ecuaţiile (3.30) şi (3.31) am utilizat următoarele notaţii:

t_C -ciclul instrucţiunii

t_{FI} -timp de execuţie a microrutinei ***Fetch* Instrucţiune** (apare la toate instrucţiunile).

t_{FO} -timp de execuţie a microrutinei ***Fetch* Operand** (apare la instrucţiunile din clasa IOP).

t_{EX} -timp de execuţie a microrutinei de **Execuţie** (apare la toate instrucţiunile).

t_{leg1} -timp de stabilire a legăturii între microrutina ***Fetch* Instrucţiune** şi cea de ***Fetch* Operand** (apare la instrucţiunile din clasa IOP).

t_{leg2} -timp de stabilire a legăturii între microrutina ***Fetch* Operand** şi cea de **Execuţie** (apare la instrucţiunile din clasa IOP).

t_{leg3} -timp de stabilire a legăturii între microrutina ***Fetch* Instrucţiune** şi cea de **Execuţie** (apare la instrucţiunile din clasa ID).

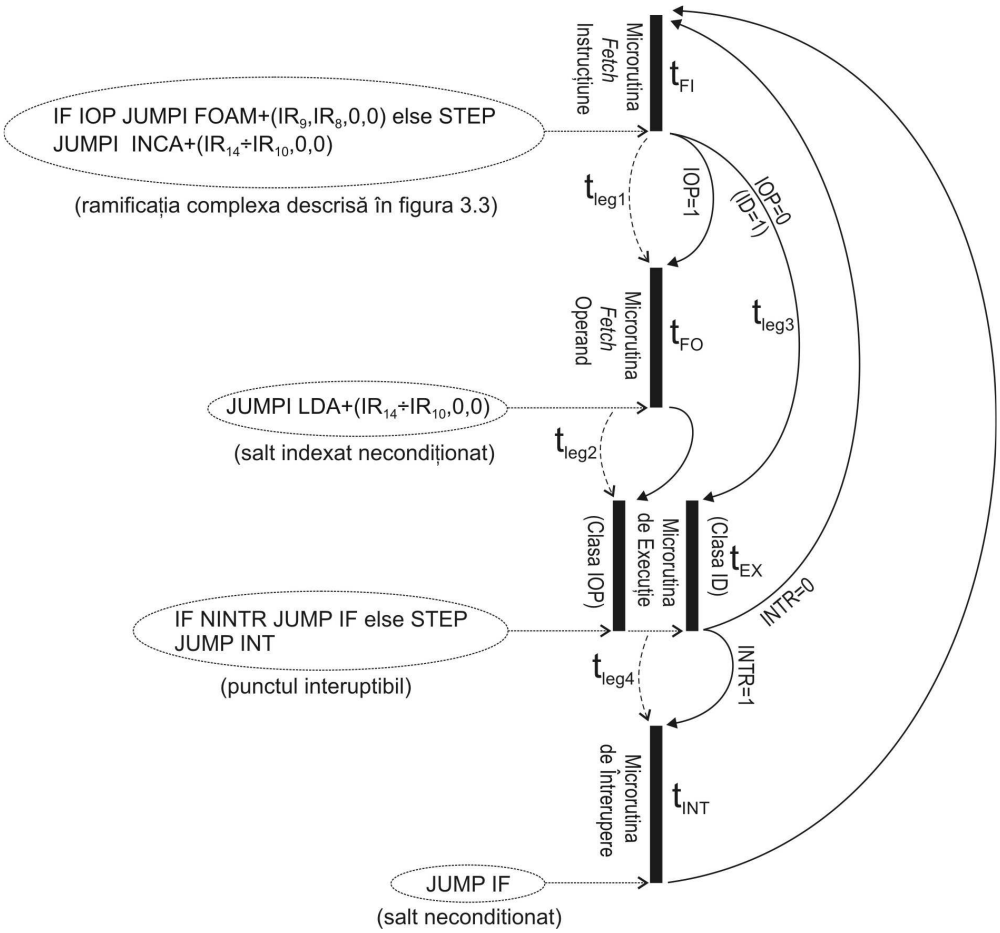


Fig 3.43 Succesiunea microrutinelor în timpul emulării instrucţiunilor

La activarea unei întreruperi, conform cu notaţiile adoptate în figura 3.43, intervin timpii:

t_{leg4} -timp de stabilire a legăturii între microrutina de **Execuţie** şi cea de **Întrerupere**.

t_{INT} -timp de execuţie a microrutinei de **Întrerupere**.

Ţelul principal al proiectantului este desigur obţinerea unui procesor microprogramat cât mai performant (cât mai rapid). Pentru atingerea acstui deziderat trebuie acţionat, concertat şi corelat, pe două direcţii:

1. Microrutine cât mai scurte (minimizarea timpului de execuţie a microrutinelor)

2. Legături cât mai rapide. (minimizarea timpilor de stabilire a legăturilor dintre microrutine)

Prima direcţie este influenţată esenţial de performanţa microinstrucţiunii. O microinstrucţiune performantă se obţine prin optimizarea formatului acesteia; criteriul acestei optimizări trebuie să fie reducerea lungimii microrutinelor din cadrul microprogramului de emulare.

Este evident că, maximizarea performanţelor microinstrucţiunii se poate obţine prin maximizarea numărului de operaţii elementare pe care microinstrucţiunea le poate declanşa şi controla. Operaţiile elementare lansate de o microinstrucţiune sunt desigur operaţii concurente şi trebuie evitate conflictele care ar putea să apară între două sau mai multe operaţii concurente.

Prin urmare, trebuie adoptat acel format de microinstrucţiune care nu introduce nici o restricţie referitoare la setul de operaţii concurente pe care proiectantul de microcod ar dori să-l activeze la un moment dat; restricţiile acceptabile sunt doar cele impuse de structura *hardware* a procesorului controlat, nu şi cele care provin din formatul microinstrucţiunii! O analiză mai amplă dedicată acestui obiectiv va fi prezentată în paragraful 3.12.

A doua direcţie este influenţată esenţial de performanţa microinstrucţiunii de ramificaţie. La 3.4.1.B am evidenţiat utilitatea salturilor indexate (utilitatea succesorilor JUMPI). De altfel, succesorii JUMPI au fost introduşi tocmai în ideia minimizării timpilor de legătură dintre microrutine. O analiză mai amplă dedicată acestui obiectiv este prezentată în paragraful următor.

3.11 Minimizarea timpilor de legătură dintre microrutine

Legăturile între microrutine se stabilesc prin microinstrucţiuni de ramificaţie. Timpul de stabilire a unei anumite legături depinde de procedeul utilizat pentru stabilirea acesteia şi de performanţele microinstrucţiunii de ramificaţie. Pentru a sintetiza cât mai bine această problemă de optimizare, precum şi soluţiile posibile, vom discuta aceşti factori pe cazul concret al timpului t_{leg1}. Timpul t_{leg1} apare numai la instrucţiunile din clasa IOP (vezi ecuaţia (3.30) cu explicaţiile aferente). Prin urmare, în analiza care urmează vom face abstracţie de instrucţiunile din clasa ID.

Legătura între microrutina ***Fetch* Instrucţiune** şi microrutinele ***Fetch* Operand** se poate stabili utilizând următoarele tehnici, respectiv microinstrucţiuni de ramificaţie:

3.11.1 Salturi simple condiţionate

Această variantă de implementare a legăturii utilizează microinstrucţiuni de ramificaţie cu succesori JUMP condiţionaţi. Modul de implementare a legăturii într-o astfel de variantă este ilustrat în figura 3.44. Analizând mecanismul prezentat în figura 3.44 constatăm că, legătura între microrutina ***Fetch* Instrucţiune** şi cele patru microrutine ***Fetch* Operand** se stabileşte:

- prin parcurgerea unei singure microinstrucţiuni de salt, în cazul AM,
- prin parcurgerea a două microinstrucţiuni de salt (succesive), în cazul AD
- prin parcurgerea a trei microinstrucţiuni de salt (succesive), în cazul AI
- prin parcurgerea a patru microinstrucţiuni de salt (succesive), în cazul AX

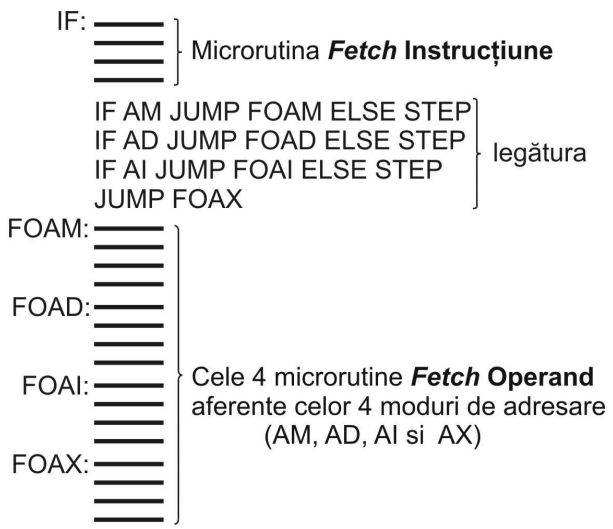


Fig 3.44 Implementarea legăturii între microrutina ***Fetch* Instrucţiune** şi cele patru microrutine ***Fetch* Operand** prin salturi simple condiţionate

Considerând ca fiind echiprobabilă apariţia celor patru cazuri (considerând că, în cadrul programelor, cele 4 moduri de adresare prezintă rate identice de utilizare), timpul mediu de stabilire a legăturii este dat de media aritmetică:

$$t_{leg1} = \frac{t_{leg1AM} + t_{leg1AD} + t_{leg1AI} + t_{leg1AX}}{4} = \frac{1+2+3+4}{4} = 2{,}5 \text{ microinstructiuni} \quad (3.32)$$

Dacă dimpotrivă, cele patru moduri de adresare au rate diferite de utilizare în cadrul programelor, atunci trebuie aplicată media aritmetică ponderată. În cazul mediei ponderate, parametrii t_{leg1AM}, t_{leg1AD}, t_{leg1AI} şi t_{leg1AX} din ecuaţia (3.32), trebuie înmulţiţi cu coeficienţii de ponderare aferenţi. Cei patru coeficienţi de ponderare se obţin evaluând frecvenţa de apariţie a celor patru moduri de adresare în cadrul celor mai uzuale programe.

3.11.2 Saltul indexat JUMPI μADR+(IR_9, IR_8)

Această legătură este redată în figura 3.45. Microinstrucţiuea de salt indexat JUMPI μADR+(IR_9, IR_8) utilizează ca INDEX biţii IR_9 şi IR_8 din codul instrucţiunii curente (câmpul MOD DE ADRESARE).

INDEX-ul utilizat poate lua patru valori: 0, 1, 2 şi 3. Prin urmare, în funcţie de valoarea INDEX-ului, această microinstrucţiune poate efectua saltul la patru microadrese succesive:

- μADR -dacă INDEX-ul are valoarea zero (IR_9,IR_8=0,0)
- μADR+1 -dacă INDEX-ul are valoarea unu (IR_9,IR_8=0,1)

- μADR+2 -dacă INDEX-ul are valoarea doi (IR_9,IR_8=1,0)
- μADR+3 -dacă INDEX-ul are valoarea trei (IR_9,IR_8=1,1)

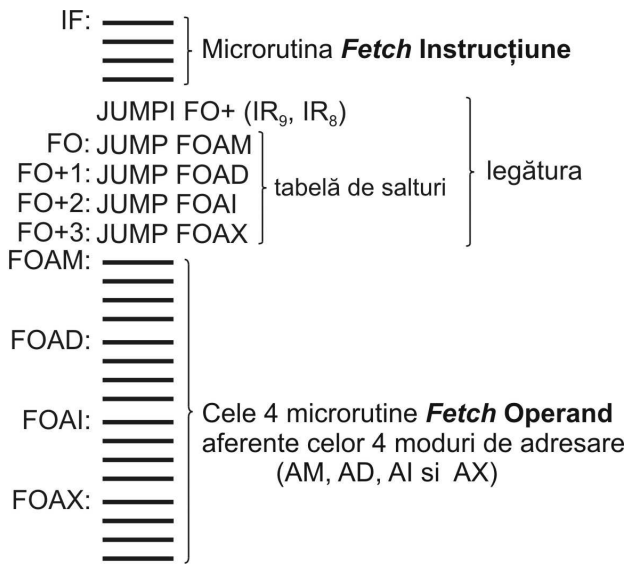


Fig 3.45 Implementarea legăturii între microrutina ***Fetch*** **Instrucţiune** şi cele patru microrutine ***Fetch*** **Operand** prin salturi indexate de forma JUMPI μADR+(IR_9, IR_8)

În cele patru locaţii MPM succesive va fi stocată o tabelă de salturi, metoda fiind ilustrată în figura 3.45. În această variantă de implementare, timpul de stabilire a legăturii între microrutina ***Fetch*** **Instrucţiune** şi microrutinele ***Fetch*** **Operand** (pe care îl vom nota cu t'_{leg1}) este acelaşi pentru toate modurile de adresare:

$$t'_{leg1} = 2 \text{ microinstrucţiuni} < t_{leg1} \qquad (3.33)$$

Soluţia este mai performantă deoarece prin această metodă am obţinut o legătură mai rapidă decât prin metoda precedentă.

3.11.3 Saltul indexat JUMPI μADR+(IR_9, IR_8, 0, 0)

Acest tip de legătură, explicitat în figura 3.46, a fost implementat şi în microprogramul de emulare din tabelul 3.16 (vezi microinstrucţiunea de la microadresa 004_H).

Microinstrucţiuea de salt indexat JUMPI μADR+(IR_9, IR_8, 0, 0) utilizează un INDEX prelucrat. Acest INDEX prelucrat conţine câmpul MOD DE ADRESARE din

codul instrucţiunii curente, la dreapta căruia se concatenează două zerouri (biţii IR_9 şi IR_8 urmaţi de doi biţi de zero). INDEX-ul utilizat poate lua patru valori: 0, 4, 8 şi 12. Prin urmare, în funcţie de valoarea INDEX-ului, această microinstrucţiune poate efectua saltul la următoarele patru microadrese:

- μADR -dacă INDEX-ul are valoarea zero (IR_9,IR_8=0,0)
- μADR+4 -dacă INDEX-ul are valoarea patru (IR_9,IR_8=0,1)
- μADR+8 -dacă INDEX-ul are valoarea opt (IR_9,IR_8=1,0)
- μADR+12 -dacă INDEX-ul are valoarea doisprezece (IR_9,IR_8=1,1)

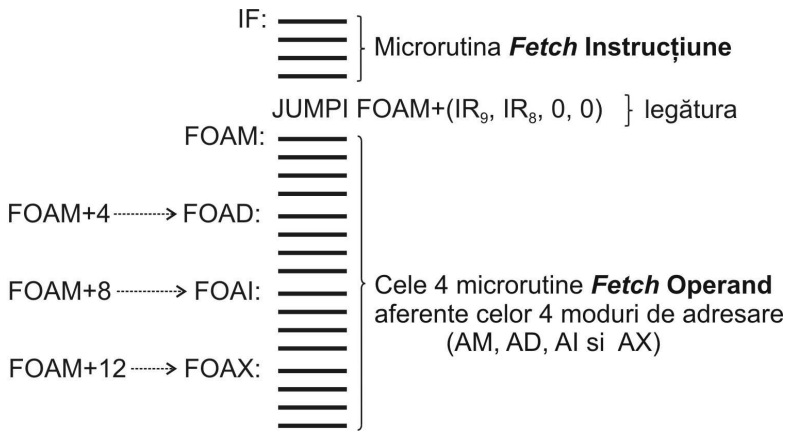


Fig 3.46 Implementarea legăturii între microrutina ***Fetch* Instrucţiune** şi cele patru microrutine ***Fetch* Operand** prin salturi indexate de forma JUMPI μADR+(IR_9, IR_8, 0, 0)

În această variantă de implementare, timpul de stabilire a legăturii între microrutina ***Fetch* Instrucţiune** şi microrutinele ***Fetch* Operand** (pe care îl vom nota cu t''_{leg1}) este acelaşi pentru toate modurile de adresare:

$$t''_{leg1} = 1 \text{ microinstrucţiune} < t'_{leg1} < t_{leg1} \qquad (3.34)$$

Prin faptul că, în structura INDEX-ului, am concatenat cei 2 biţi de zero la dreapta câmpului MOD DE ADRESARE, am rezervat câte 4 locaţii MPM (4 microinstrucţiuni) pentru fiecare microrutină ***Fetch* Operand**. Numărul de locaţii MPM (numărul de microinstrucţiuni) rezervate a fost stabilit după proiectarea celor patru microrutine ***Fetch* Operand**. Constatând că lungimea medie a celor patru microrutine se situează sub numărul de patru microinstrucţiuni, am decis să adăugăm doi biţi de zero în structura INDEX-ului. Dacă lungimea medie a celor patru microrutine s-ar fi situat în jurul numărului de două

microinstrucţiuni, ar fi fost oportună adăugarea unui singur bit de zero în structura INDEX-ului (salt din două în două microinstrucţiuni).

Decizia de a implementa salturi indexate în cadrul microinstrucţiunii de ramificaţie (salturi care conduc la creşterea complexităţii unităţii de control microprogramate) o justificăm deci prin obiectivul pe care îl urmărim cu asiduitate: maximizarea performanţelor.

3.12 Optimizarea formatului microinstrucţiunii

Optimizarea formatului microinstrucţiunii are drept scop minimizarea timpului de execuţie a microrutinelor (obiectiv realizabil prin minimizarea lungimii microrutinelor).

Cea mai performantă microinstrucţiune este aceea care permite proiectantului să activeze întreg setul de comenzi elementare dorit, în fiecare punct al microprogramului de emulare. Aşa cum afirmam la finele paragrafului 3.10, trebuie adoptat acel format de microinstrucţiune care nu introduce nici o restricţie referitoare la setul de operaţii concurente pe care proiectantul de microcod ar dori să-l activeze la un moment dat; restricţiile acceptabile sunt doar cele impuse de structura *hardware* a procesorului controlat, nu şi cele care provin din formatul microinstrucţiunii!

În paragraful 3.4 am proiectat microinstrucţiunea aferentă procesorului didactic. Am decis să implementăm trei tipuri de microinstructiuni (de ramificaţie, de transfer sincron şi de transfer asincron) şi am adoptat formatul fiecărui tip în parte. Fiecare tip de microinstrucţiune a fost dedicat unui anumit scop:

- Microinstrucţiunea de ramificaţie pentru implementarea legăturilor dintre microrutine

- Microinstrucţiunea de transfer sincron pentru controlul transferurilor de date între registrele interne aferente procesorului (cu eventuala procesare în ALU a informaţiilor transferate).

- Microinstrucţiunea de transfer asincron pentru controlul transferurilor procesor-memorie (controlul operaţiilor de citire din şi respectiv de scriere în memorie).

Trebuie să menţionăm faptul că această decizie a avut la bază raţiuni didactice şi nu de performanţă. Luând decizia de a proiecta trei tipuri diferite de microinstrucţiuni (fiecare tip fiind dedicat unui anumit scop), este evident că ne-am asumat conştient riscul de a „porni la drum” cu o microinstrucţiune neoptimizată. Altfel spus, cu trei microinstrucţiuni dedicate, care controlează tipuri diferite de operaţii, am acceptat premisa unei „organizări pe verticală” a microprogramului de emulare; trei operaţii diferite (un transfer sincron, un transfer asincron şi o ramificaţie în microcod) nu por fi codificate decât în microinstrucţiuni diferite, care nu pot fi procesate decât succesiv (microprogramare pe verticală) şi nu concomitent (microprogramare pe orizontală). Prin urmare, decizia de a proiecta trei tipuri diferite de microinstrucţiuni, conduce inevitabil la microrutine mai lungi iar microrutinele lungi vor genera un procesor mai lent.

Pentru a putea procesa concurent cele trei tipuri de operaţii (un transfer sincron, un transfer asincron şi o ramificaţie) trebuie să plecăm de la premisa unei microinstrucţiuni generale, capabilă să controleze toate aceste operaţii. În această manieră, într-o singură microinstrucţiune se va putea codifica un set amplu de microcomenzi, care va iniţia şi

executa (concurent) un transfer sincron, un transfer asincron şi o ramificaţie în microcod. Evident că prin această paralelizare a operaţiilor controlate se va ajunge de fapt la o reducere a lungimii microrutinelor din cadrul microprogramului de emulare.

Microinstrucţiunea generală va fi mai performantă dar va avea o lungime mai mare. Vom încerca să introducem acest concept de microinstrucţiune generală pornind de la cele trei tipuri de microinstrucţiuni proiectate la 3.4. Paşii care ar trebui urmaţi pentru proiectarea microinstrucţiunii generale sunt:

1. Concatenarea microinstrucţiuniilor de transfer sincron (pe 20 biţi), de transfer asincron (pe 20 biţi) şi de ramificaţie (pe 20 biţi) într-o singură microinstrucţiune generală (pe 60 biţi).

2. Analiza microinstrucţiunii generale (obţinută în pasul 1) în scopul de a identifica biţii şi respectiv câmpurile care îşi pierd semnificaţia (în cadrul microinstrucţiunii generale).

3. Eliminarea biţilor (câmpurilor) care îşi pierd semnificaţia

4. Rafinarea formatului de microinstrucţiune generală (obţinut în pasul 3) în vederea optimizării acestuia.

Paşi 1, 2 şi 3 sunt sintetizaţi în figura 3.47, în două variante de lucru.

Pasul 1:
Pasul 1 este identic în cele două variante: se concatenează microinstrucţiunile de transfer sincron, de transfer asincron şi de ramificaţie definite la 3.4. Prin concatenare se obţine un cuvânt binar, pe 60 biţi, pe care îl vom denumi microinstrucţiune generală (acceptând în mod deliberat un mic abuz de limbaj). În această microinstrucţiune generală se regăsesc (în ordine) toate câmpurile definite în cadrul microinstrucţiunilor de transfer sincron, de transfer asincron şi de ramificaţie.

În pasul 2 urmează să fie identificaţi biţii şi respectiv câmpurile care îşi pierd semnificaţia iar în pasul 3 vor fi eliminaţi aceşti biţi şi respectiv aceste câmpuri.

Pasul 2:
Analiza pe care o facem în acest pas cu scopul de a identifica biţii şi respectiv câmpurile care îşi pierd semnificaţia conduce la următoarele concluzii:

- Biţii:

  -MIR_{19} şi MIR_{18} din cadrul microinstrucţiunii de transfer sincron

  -MIR_{19} şi MIR_{18} din cadrul microinstrucţiunii de transfer asincron

  -MIR_{19} din cadrul microinstrucţiunii de ramificaţie

  îşi pierd semnificaţia în cadrul microinstrucţiunii generale şi prin urmare, în pasul 3 vor fi eliminaţi. **Aceşti 5 biţi sunt eliminaţi în ambele variante de lucru (vezi figura 3.47)**.

Argumentaţia care stă la baza acestei decizii:
În cadrul celor trei tipuri de microinstrucţiuni (de transfer sincron, de transfer asincron şi de ramificaţie), bitul MIR_{19} specifică tipul microinstrucţiuni:

$MIR_{19}=0$ -defineşte microinstrucţiunea de ramificaţie

$MIR_{19}=1$ -defineşte microinstrucţiunea de transfer (sincron sau asincron)

În cazul microinstrucţiunilor de transfer, următorul bit (MIR_{18}) defineşte tipul microinstrucţiunii de transfer:

$MIR_{18}=0$ -defineşte microinstrucţiunea de transfer sincron

$MIR_{18}=1$ - defineşte microinstrucţiunea de transfer asincron

Microinstrucţiune generală înseamnă în fapt un singur tip de microinstrucţiune şi prin urmare nu mai sunt necesari aceşti biţi de clasificare şi respectiv identificare a celor trei tipuri diferite de microinstrucţiuni.

- Biţii $MIR_{11\div8}$ din cadrul microinstrucţiunii de ramificaţie (cei mai semnificativi 4 biţi din câmpul MICROADRESA DE SALT) nu sunt necesari şi prin urmare, în pasul 3 vor fi eliminaţi.

Argumentaţia care stă la baza acestei decizii *a fost prezentată la 3.9 (vezi figura 3.41 şi explicaţiile aferente). Aceşti 4 biţi sunt eliminaţi în ambele variante de lucru (vezi figura 3.47).*

- Câmpurile SURSA SBUS, SURSA DBUS, OPERAŢIE ALU şi SURSA RBUS apar atât în microinstrucţiunea de transfer sincron cât şi în cea de transfer asincron.

  În varianta 1 de lucru, în pasul 3, vor fi integrate în microinstrucţiunea generală doar cele patru câmpuri care provin din microinstrucţiunea de transfer sincron; cele care provin din microinstrucţiunea de transfer asincron vor fi eliminate.

  În varianta 2 de lucru, în pasul 3, vor fi integrate în microinstrucţiunea generală ambele seturi de patru câmpuri. Pentru diferenţiere, cele patru câmpuri care provin din microinstrucţiunea de transfer sincron le vom eticheta cu indicele „S” (SURSA $SBUS_S$, SURSA $DBUS_S$, OPERAŢIE ALU_S şi SURSA $RBUS_S$) iar cele care provin din microinstrucţiunea de transfer asincron le vom eticheta cu indicele „A” (SURSA $SBUS_A$, SURSA $DBUS_A$, OPERAŢIE ALU_A şi SURSA $RBUS_A$).

Să reamintim faptul că în cele patru câmpuri pot fi codificate patru microcomenzi prin care:

- în cadrul microinstrucţiunii de transfer sincron se selectează registrele sursă, se controlează operaţia ALU şi se selectează rezultatul ALU pentru a fi emis pe RBUS. Microcomenzile codificate în aceste patru

câmpuri, împreună cu o microcomandă de tip *„Primeşte Registru”* codificată în câmpul DESTINAŢIE RBUS, vor controla de fapt transferul sincron din interiorul procesorului (transfer registru-registru cu eventuala procesare în ALU a informaţiilor transferate). Microcomenzile codificate în aceste câmpuri sunt activate în starea ST_1 a automatului SEQ (vezi figura 3.32 cu explicaţiile aferente).

- în cadrul microinstrucţiunii de transfer asincron, cele patru câmpuri în discuţie au semnificaţie numai dacă în câmpul MEMORY OPERATION este codificată microcomanda WRITE; microcomenzile codificate în cele patru câmpuri vor fi activate în satrea ST_3 a automatului SEQ şi au rolul de a emite pe RBUS (MBUS) datele de scris în memorie (vezi figura 3.32 cu explicaţiile aferente).

Argumentaţia care stă la baza deciziei din cadrul variantei 1 de lucru:
Microcomenzile codificate în cele patru câmpuri, în starea ST_1 a automatului SEQ, vor emite pe RBUS datele ce vor fi implicate în transferul sincron. Aceleaşi microcomenzi, reactivate pe durata starii ST_3 a automatului SEQ, vor emite pe RBUS (MBUS) datele ce vor fi scrise în memorie prin microcomanda WRITE. Prin urmare, în varianta 1 de lucru, datele ce vor fi încărcate în registrul destinaţie (transferul sincron) şi datele ce vor fi memorate în memorie prin microcomanda WRITE (transferul asincron) nu pot fi decât identice (constrângere acceptată în mod deliberat).

Argumentaţia care stă la baza deciziei din cadrul variantei 2 de lucru:
Microcomenzile cu indicele „S” (SURSA $SBUS_S$, SURSA $DBUS_S$, OPERAŢIE ALU_S şi SURSA $RBUS_S$) vor fi decodficate de un set de patru decodificatoare care vor fi activate pe durata stării ST_1 a automatului SEQ. Microcomenzile cu indicele „A” (SURSA $SBUS_A$, SURSA $DBUS_A$, OPERAŢIE ALU_A şi SURSA $RBUS_A$) vor fi decodficate cu ajutorul unui alt set de patru decodificatoare care vor fi activate pe durata stării ST_3 a automatului SEQ. Prin urmare, în această variantă de lucru, datele ce vor fi încărcate în registrul destinaţie (transferul sincron) pot fi altele decât datele ce vor fi memorate în memorie prin microcomanda WRITE (transferul asincron).

În concluzie, microinstrucţiunea generală obţinută în varianta 2 de lucru este mai performantă decât cea obţinută în varianta 1. Acest plus de performanţă are un cost: lungimea microinstrucţiunii mai mare cu 3×4=12 biţi, ceea ce conduce la o creştere cu 12 biţi a dimensiunii pe orizontală a planului de memorie MPM. Plusul de performanţă poate fi însă fructificat numai pe microinstrucţiuni care fac scriere în memorie. Astfel de microinstrucţiuni apar în microprogramul de emulare doar în anumite microrutine de execuţie (la instrucţiunile care specifică un operand *dest* şi nu unul *src* şi doar în cazul modurilor de adresare AI şi AX).

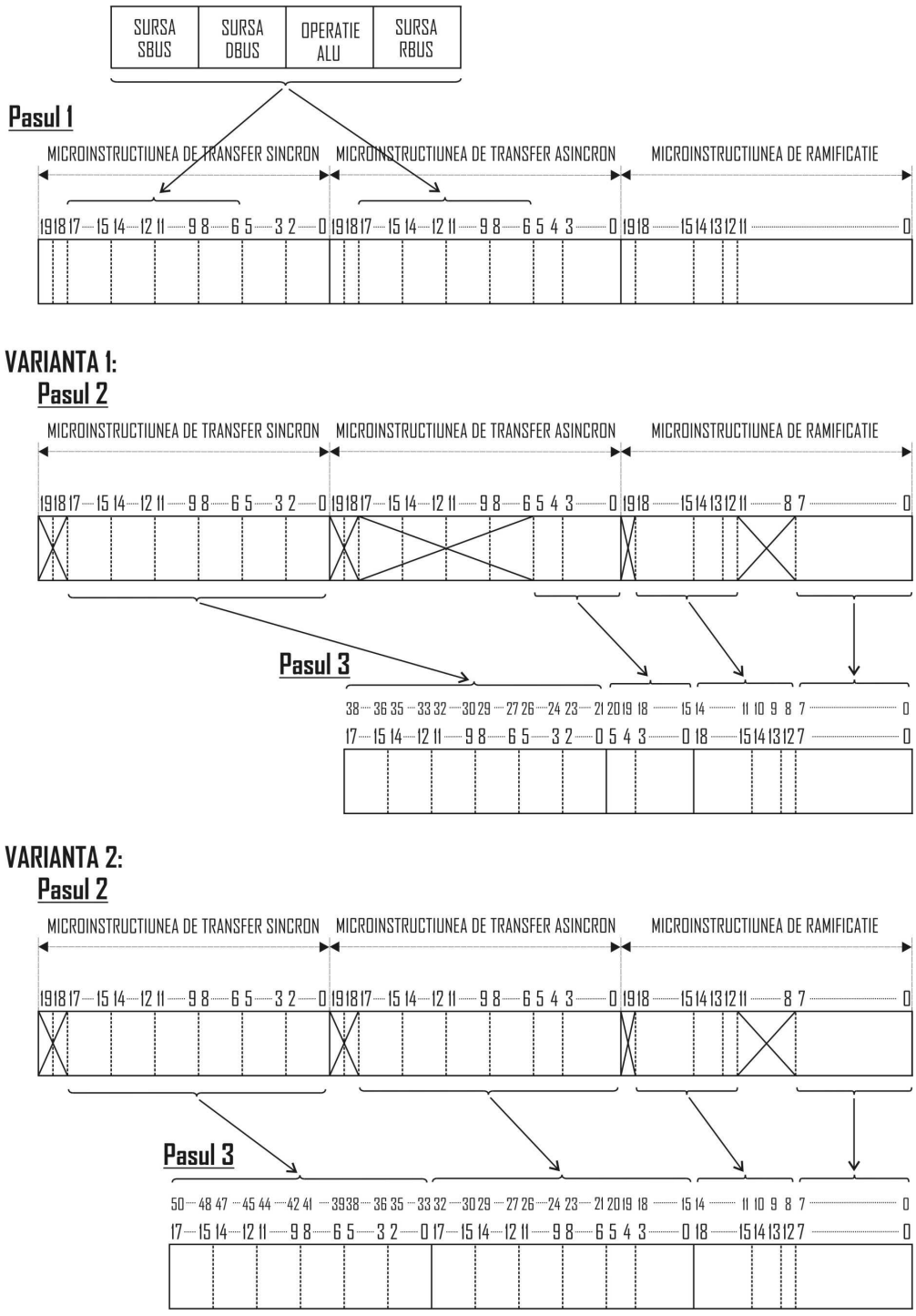


Fig 3.47 Proiectarea microinstrucţiunii generale

Pentru a alege varianta optimă de lucru proiectantul trebuie să facă o evaluare destul de complicată. Trebuie evaluat plusul de performanţă pe care microinstrucţiunea generală obţinută în varianta 2 de lucru îl aduce în raport cu microinstrucţiunea rezultată în varianta 1. Pentru aceasta trebuie să evalueze (statistic) frecvenţa instrucţiunilor care fac scriere în memorie în cadrul microrutinei de **Execuţie** şi să calculeze plusul de performanţă obtenabil în varianta 2 de lucru. Acestui plus de performanţă trebuie să i se contrapună costul suplimentar reprezentat de *hardware*-ul adiţional necesar în varianta 2 de lucru.

Pasul 4 presupune rafinarea microinstrucţiunii obţinute în pasul anterior. Rafinarea presupune un proces de analiză care utilizează, ca date de intrare, microprogramul de emulare. Să dăm un exemplu de problemă care trebuie soluţionată în această etapă:

În microinstrucţiunea generală obţinută la finele pasului 3 (în ambele variante de lucru) vom avea un câmp OTHER OPERATION (provenit din microinstrucţiunea de transfer sincron) şi un câmp SHIFT&OTHER OPERATION (provenit din microinstrucţiunea de transfer asincron). Ambele sunt câmpuri de tip *„Other Operations”* pentru că conţin microcomenzi din această categorie. Întrebarea care se pune este:

În microinstrucţiunea generală vom implementa într-adevăr două câmpuri de tip „Other Operations”sau vom reuni cele două câmpuri într-unul singur ?

Prin reunire am face economie la nivelul *hardware*-ului pentru că prin reunire lungimea microinstrucţiunii scade (”microprogramare pe verticală”). Pe de altă parte, într-o microinstrucţiune cu un singur câmp *„Other Operations”* vom putea codifica o singură microcomandă din categoria „*other*”. Într-o microinstrucţiune cu două câmpuri de tip *„Other Operations”* (*„Other Operations1”* şi respectiv *„Other Operations2”)* vom putea codifica două microcomenzi de tip „*other*”, simultane (concurente).

Concluzia: Din nou trebuie făcută o evaluare a microinstrucţiunilor din cadrul microprogramului de emulare şi inventariate situaţiile în care ar trebui activate concurent (din aceeaşi microinstrucţiune) mai multe microcomenzi de tip „*other*”. Pe baza acestei evaluări vom obţine răspunsul la întrebarea: „câte câmpuri de tip *„Other Operations”* vom implementa în microinstrucţiunea generală ?”.
Răspunsul ar putea fi: „unul, două sau chiar mai multe câmpuri de acest tip”.

În final, să subliniem faptul că, la proiectarea unui procesor microprogramat, rafinarea formatului microinstrucţiunii se realizează printr-un proces iterativ. La prima iteraţie se schiţează un format de microinstrucţiune. Nu acesta va fi formatul final deoarece nu se vor putea stabili cu exactitate, în această etapă, toate câmpurile şi respectiv microcomenzile care trebuie codificate în aceste câmpuri. De exemplu, nu se cunosc cu exactitate indecşii care vor fi utilizaţi în microprogram. Nu se ştie câte zerouri vor fi concatenate la dreapta biţilor unui anumit INDEX pentru că nu a fost încă evaluată lungimea microrutinelor vizate de acel INDEX. Nu se cunosc exact condiţiile care trebuie testate în microprogram (în microinstrucţiunile de salt condiţionat). Doar după elaborarea microprogramului (într-o primă variantă) se poate obţine o listă a condiţiilor testate şi pe baza acestei liste se va putea definitiva câmpul SUCCESOR din formatul microinstrucţiunii.

Pe baza acestui format de microinstrucţiune (în schiţă) se trece la elaborarea microprogramului de emulare. După elaborarea microprogramului se va reveni la formatul microinstrucţiunii pentru rafinarea lui. Procesul de rafinare presupune multiple evaluări şi poate chiar multiple iteraţii.

Microinstrucţiunea generală poate controla multiple operaţii paralele:

- din zona de transfer sincron poate controla un transfer registru-registru (cu eventuala procesare în ALU a informaţiilor transferate)
- din zona de transfer asincron poate controla un transfer procesor-memorie (citire din memorie sau scriere în memorie)
- din zona de ramificaţie poate specifica orice succesor.

Datorită potenţialului ridicat de paralelism pe care îl prezintă, microinstrucţiunea generală va conduce la reducerea lungimii microrutinelor din cadrul microprogramului de emulare. Cu cât lungimea microrutinelor scade, cu atât performanţele procesorului cresc.

Vom ilustra plusul de performanţă pe care microinstrucţiune generală îl poate aduce printr-un exemplu. Primele trei microinstrucţiuni din cadrul microprogramului de emulare aferent procesorului didactic (tabelul 3.16) realizează *fetch*-ul instrucţiunii din memorie. Acestea sunt:

```
NONE DBUS PC, PdALU, PmADR          ; ADR←PC  (încărcarea adresei de fetch în
                                    ; registrul ADR)
IFCH, +2PC                          ; IR←MEM|ADR (fetch instrucţiune) şi
                                    ; PC←PC+2 (incrementare PC)
IF ACLOW JUMP PWFAIL else STEP      ; test ACLOW  (punct de ramificaţie în 2 direcţii)
```

Prima este o microinstrucţiune de transfer sincron, a doua de transfer asincron şi a treia de ramificaţie. În cazul aplicării formatului general, aceste trei microinstrucţiuni pot fi colapsate într-una singură de forma:

```
NONE DBUS PC, PdALU, PmADR, IFCH, +2PC, IF ACLOW JUMP PWFAIL else STEP
```

Această microinstrucţiune generală activează următoarele microcomenzi:

NONE	-în câmpul SURSA SBUS
PdPC	-în câmpul SURSA DBUS
DBUS	-în câmpul OPERAŢIE ALU
PdALU	-în câmpul SURSA RBUS
PmADR	-în câmpul DESTINAŢIE RBUS
IFCH	-în câmpul MEMORY OPERATION
+2PC	-în câmpul SHIFT&OTHER OPERATIONS

Primele cinci microcomenzi vor fi activate în starea ST_1 a automatului SEQ (provin din zona de transfer sincron a microinstrucţiunii generale) şi prin urmare vor realiza operaţia ADR←PC. Următoarele două (IFCH şi +2PC) provin din zona de transfer asincron a microinstrucţiunii generale şi vor realiza operaţiile PC←PC+2 (în starea ST_1 a automatului SEQ) şi respectiv IR←MEM$|_{ADR}$ (*fetch*-ul instrucţiunii), în starea ST_3 a automatului SEQ. Prin urmare, mai întâi se transferă adresa de *fetch* din registrul PC în registrul ADR (în starea ST_1) şi mai apoi se extrage codul instrucţiunii din memorie (în starea ST_3); este exact succesiunea corectă a operaţiilor.

Comanda +2PC este activată tot în starea ST_1 a automatului SEQ. Dacă vom proiecta o microcomandă +2PC de tip impuls atunci frontul activ (urcător) al acestei microcomenzi va fi sincron cu frontul activ al microcomenzii PmADR (care este tot de tip impuls). În aceste condiţii, în registrul ADR se va încărca valoarea existentă în registrul PC înaintea operaţiei de incrementare. Altfel spus, mai întâi se face operaţia ADR←PC şi apoi operaţia PC←PC+2; este exact succesiunea corectă a operaţiilor.

Suplimentar, în starea ST_1 a automatului SEQ, microinstrucţiunea generală impune şi execuţia succesorului condiţionat:

IF ACLOW JUMP PWFAIL else STEP.

Pentru execuţia acestui succesor dublu, în starea ST_1, automatul SEQ activează comanda:

- LdMAR, dacă ACLOW=1 (se execută succesorul JUMP PWFAIL)
- +1MAR, dacă ACLOW=0 (se execută succesorul STEP)

Cele trei microinstrucţiuni pe care am construit exemplul de colapsare fac parte din microrutina ***Fetch* Instrucţiune** (tabelul 3.16). Prin colapsare (raport 3:1) lungimea microrutinei ***Fetch* Instrucţiune** se reduce semnificativ şi deci performanţa procesorului creşte. Evident că, la nivelul întregului microprogram de emulare, colapsarea nu se va putea face în raportul de trei la unu deoarece vor fi şi situaţii în care colapsarea nu este posibilă sau în care este parţial posibilă.

Pentru a o face însă posibilă, trebuie introdusă o microinstrucţiune generală, capabilă să controleze, multiple operaţii concurente:

- transferuri registru-registru (interne procesorului), cu eventuala procesare în ALU a informaţiilor transferate
- transferuri procesor-memorie (citire din şi respectiv scriere în memorie)
- operaţii din caregoria „*other*” (interne procesorului)
- ramificaţii în microcod (succesori STEP, JUMP şi JUMPI, condiţionaţi şi necondiţionaţi)

Un lucru rămâne cert: prin introducerea microinstrucţiunii generale performanţele procesorului microprogramat cresc semnificativ.

Microprogramare. Proiectare avansată

În capitolul 3 am introdus conceptul microprogramării şi 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 şi variat) de instrucțiuni şi care utilizează formatul cu două adrese (vezi 1.3.2).

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

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

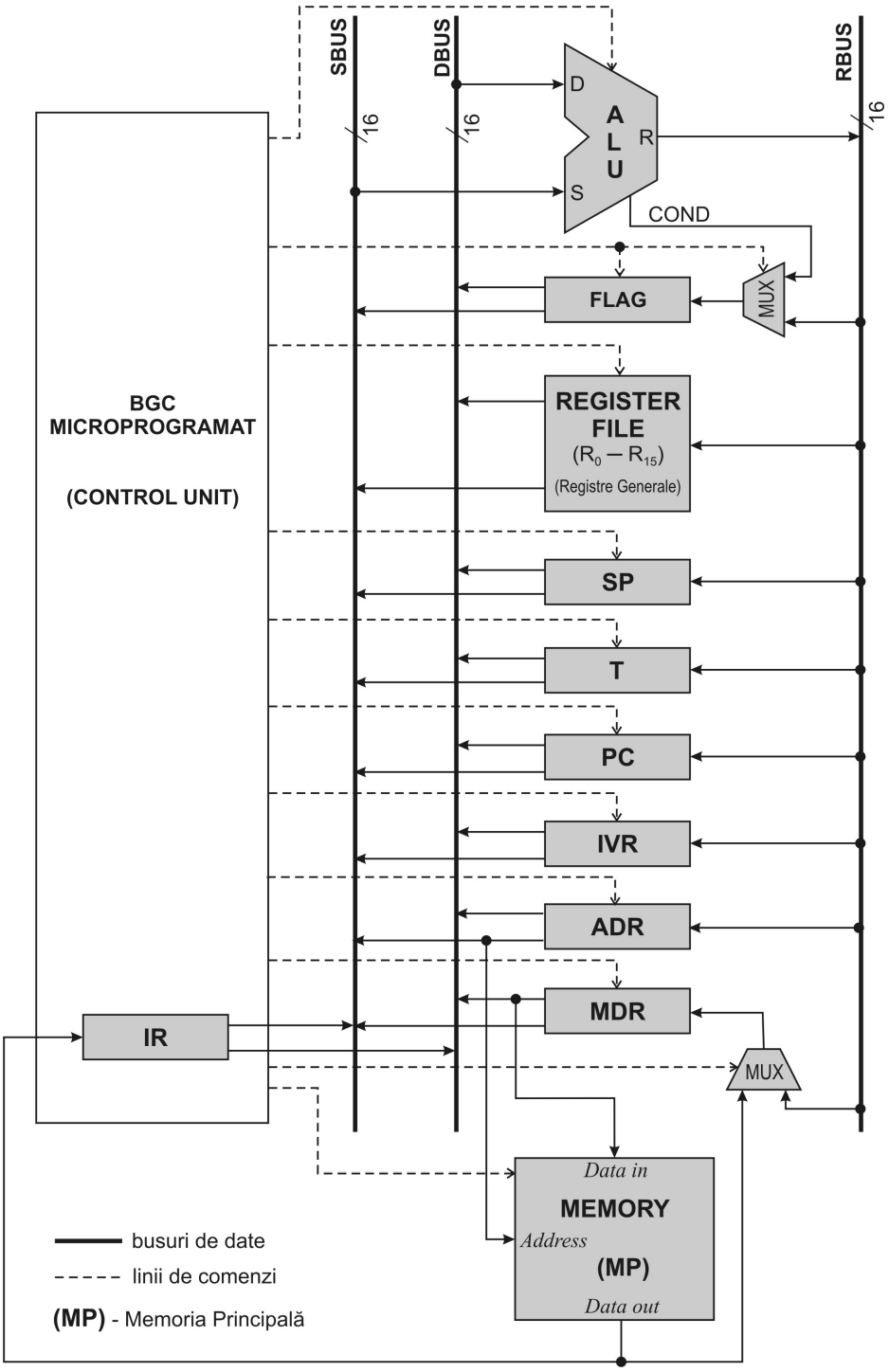


Fig 4.1 Schema bloc aferentă procesorului CISC

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

În cuncluzie, în cadrul fazei (microrutinei) de execuție aferentă instrucțiunilor aritmetice şi logice, operanzii (sursă şi respectiv destinație) vor fi emişi pe SBUS şi respectiv DBUS (prin activarea unor comenzi de tip „*Predă Registru*", 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 „*Primeşte Registru*"). Suplimentar, prin activarea unei comenzi PdCOND („*Predă Condiții*"), semnalele de condiții generate de unitatea ALU (C_{out}, Z_R, S_R şi DCR) vor fi încărcate în bistabilii de condiții (C-*Carry*, Z-*Zero*, S-*Sign* şi V-*oVerflow*) din registrul FLAG.

În structura procesorului pot fi identificate următoarele registre:

FLAG –registrul de *flag*-uri. Conține cele 4 *flag*-uri de condiții: C, Z, S şi 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 şi 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     ; R1-R2 cu poziționare flag-uri (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 şi apoi testat de instrucțiunea BNE).

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

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

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

Indicatorii (bistabilii) de condiții C, Z, S şi 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 ieşire 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 ieşirile ALU)

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

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

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

V –*overflow* (depăşire)

V=1 indică rezultat incorect (depăşirea domeniului de reprezentare)
V=0 indică rezultat corect (fără depăşire de domeniu)

Observație:
Mecanismul de ramificație bazat pe flag-uri de condiții şi 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 maşină RISC (pentru a maximiza performanțele), compilatorul trebuie să parcurgă două faze succesive: ***generare cod maşină*** *(translatare), urmată de* ***optimizarea codului maşină generat*** *(reordonarea instrucțiunilor fără a afecta logica programului). Reordonarea are drept scop eliminarea a cât mai multor dependențe de date dintre instrucțiunile programului maşină. Altfel spus, se minimizează numărul de dependențe (hazarduri) de date.*

Ramificațiile implementate pe două instrucțiuni succesive impun restricții compilatorului cu reorganizare. Dacă ne referim la exemplul de mai sus (fig 4.2), între instrucțiunile CMP şi BNE nu pot fi inserate (prin reordonare) instrucțiuni aritmetice sau logice chiar dacă acestea sunt independente (în sensul că nu operează cu regiştrii R1 şi 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 succesive este evitat la RISC.

La procesoarele RISC, exceptând instrucțiunile Load şi 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) şi din acest motiv procesorul RISC se mai numeşte şi maşină load-store.

La RISC, ramificațiile se implementează de regulă pe o singură instrucțiune care, mai întâi efectuează o operație aritmetico-logică (cu operanzi preluați din registre) pentru a evalua condiția testată şi 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 şi în cazul maşinii CISC. Maşina RISC implică însă un transfer de complexitate din hardware în software şi prin urmare, optimizarea codului la compilare este mult mai importantă decât în cazul CISC; performanța maşinii RISC (la execuția programului) este mult mai dependentă de optimizarea realizată la compilare.

```
      BNE R1,R2,L1   ; efectuează R1-R2 şi fă salt la adresa L1 dacă R1-R2≠0
      --------       ; ramura pe care se continuă (după BNE R1,R2,L1)
      --------       ; dacă condiția de salt este neândeplinită
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. Conține16 registre pe 16 biți ($R_0 \div R_{15}$), implementate sub forma unei matrici de memorie statică rapidă cu două porturi de citire şi 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 şi emis pe SBUS
- un registru (R_j; j=0÷15) citit şi 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 $R_k \leftarrow R_i$ 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 şi 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 şi 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) şi respectiv este încărcat cu datele citite din memorie în ciclurile de citire (ieşirea "*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 comenzii *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 comenzii *READ* pentru MP (memoria principală);
- încărcarea datelor (citite din MP) în MDR.

Registrele ARD şi MDR sunt deci destinate inerfațării cu memoria (pe adrese şi 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) şi 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 citeşte 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 citeşte operandul destinație din MP (în MDR);
- se adună cei doi operanzi (T+MDR) în ALU şi 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 şi cu ambii operanzi în memorie (adresare indirectă sau indexată atât la sursă cât şi la destnație), sunt necesare două registre tampon (MDR şi T) pentru a putea efectua *fetch*-ul celor doi operanzi din memorie.

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

4.1.2 Setul de instrucțiuni

Pentru proiectarea setului de instrucțiuni vom utiliza formatul cu două adrese, format uzual la procesoarele CISC (vezi 1.3.2) iar pentru localizarea operanzilor vom utiliza patru moduri de adresare (cele mai uzuale): imediat, direct, indirect şi 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** şi respectiv **operand destinație**. Pentru localizarea lor vom utiliza patru moduri de adresare: imediat, direct, indirect şi 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 priveşte 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 şi respectiv memorie-memorie.

Localizare operand sursă	Localizare operand destinație
registru	registru
registru	memorie
memorie	registru
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).

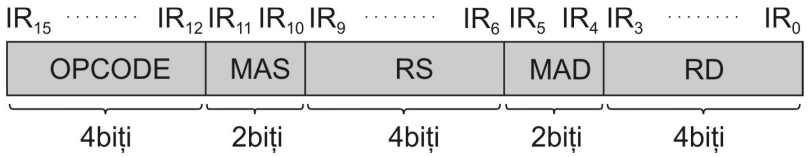


Fig 4.4 Formatul instrucțiunilor cu 2 operanzi

Semnificația câmpurilor evidențiate în formatul instrucțiunii cu 2 operanzi este:

OPCODE –*opcode*-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 –registrul 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 şi MAD sunt câmpuri de 2 biți. Procesorul conține 16 registre generale (fig 4.1) iar în câmpurile RS şi RD se codifică adresa registrului sursă şi respectiv adresa registrului destinație. Prin urmare, câmpurile RS şi 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 şi 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 depăşeşte 8, atunci va fi suficient un OPCODE pe 4 biți (un bit pentru a specifica clasa şi 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 şi 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 dest,src        ; dest←src
                    ; dest=operand destinație
                    ; src=operand sursă
```

Exemple:

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

MOV R4,(R2)         ; R4←MEM|R2 (conținutul locației de memorie
                    ; adresată de R2 este transferat în registrul R4.
                    ; Avem adresare indirectă la sursă şi adresare directă
                    ; la destinație)
```

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

- instrucțiuni aritmetice:

ADD *dest,src* ; *dest*←*dest*+*src*

SUB *dest,src* ; *dest*←*dest*–*src*

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

Exemple:

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

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

CMP R0,(R1) ; $R_0 - MEM|_{R1}$ (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*←*dest* AND *src*

OR *dest,src* ; *dest*←*dest* OR *src*

XOR *dest,src* ; *dest*←*dest* XOR *src*

Exemple:

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

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

XOR R3,8 ; $R_3 \leftarrow R_3$ XOR 8 (operația logcă XOR între registrul
; R_3 şi constanta 8, cu depunerea rezultatului în R_3)

Am utilizat următoarele notații consacrate:

- **Ri** specifică conținutul registrului R_i (adresare directă)
- **(Ri)** specifică conținutul locației de memorie adresată de R_i (adresare indirectă)
- **k(Ri)** 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 şi logice poziționează şi *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 şi S; la instrucțiunile logice nu poate să rezulte niciodată C=1 sau V=1 şi prin urmare, nu are sens poziționara acestor două *flag*-uri.

Făcând inventarul, constatăm că am definit 7 instrucțiuni cu 2 operanzi (MOV, ADD, SUB, CMP, AND, OR şi 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 îşi modifică valoarea şi prin această modificare a valorii, îşi pierde în fond statutul de constantă. Prin urmare, adresarea imediată la destinație reprezintă un nonsens şi de aceea este consideată cod ilegal (instrucțiune ilegală).

B. Instrucțiuni cu un singur operand

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

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

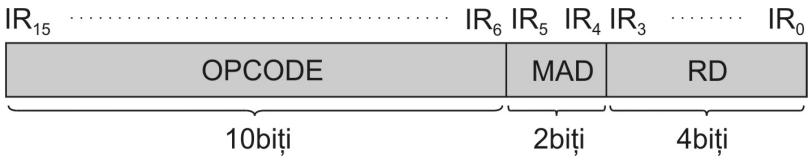


Fig 4.5 Formatul instrucțiunilor cu un singur operand

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 şi aritmetice:

  CLR *dest* ; *dest*←0

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

  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 R_0)

  NEG R3 ; $R_3 \leftarrow \overline{R}_3$ (inversează bit cu bit operandul din R_3)

  INC (R2) ; MEM|$_{R2}$←MEM|$_{R2}$+1 (incrementează cu 1
  ; conținutul locației de memorie adresată de R_2)

  DEC R5 ; $R_5 \leftarrow R_5 - 1$ (decrementează cu 1 conținutul
  ; registrului R_5)

- instrucțiuni de deplasare şi rotire aritmetică şi logică:

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

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

Exemple:

ASL R1	; deplasare aritmetică la stânga cu o poziție ; a operandului din R_1 (înmulțire cu 2)
ASR (R2)	; deplasare aritmetică la dreapta cu o poziție ; a conținutului locației de memorie adresată ; de R_2 (împărțire la 2)
LSR 14(R0)	; deplasare logică la dreapta cu o poziție ; a conținutului locației de memorie de la ; adresa R_0+14
RLC R7	; rotire la stânga cu *carry* a registrului R7

- instrucțiunile PUSH Ri şi POP Ri

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

Exemple:

PUSH R3	; stiva←R_3 (se adaugă o locație în vârful ; stivei şi în noua locație se memorează R_3)
POP R5	; R_5←stiva (conținutul locației din vârful stivei ; se transferă în R_5 şi locația se elimină din stivă)

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

Instrucțiunile aritmertice şi logice (inclusiv cele de deplasare) poziționează *flag*-urile de condiții în conformitate cu rezultatul obținut. Deplasările şi rotirile se vor efectua cu ajutorul unui bloc SHIFT, bloc combinațional din componența ALU (fig 2.59). Dacă în figura 2.60 înlocuim registrul A cu blocul SHIFT, obținem schemele de deplasare şi rotire 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 şi rotire asupra intrărilor extreme aferente blocului SHIFT (intrarea din extrema dreaptă şi respectiv intrarea din extrema stângă).

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

C. Instrucțiunile de salt

Această clasă conține instrucțiuni de salt condiționat, instrucțiunea de salt necondiționat şi 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 maşină, ramificațiile condiționate (vezi explicațiile relative la registrul FLAG de la 4.1.1). Codul maşină 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 maşină utilizând instrucțiuni de ramificație (de salt condiționat).

Construcțiile pe care le regăsim în limbajele de nivel înalt revendică 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 (complicația) de a calcula, el însuşi, adresele de salt. De exemplu, instrucțiunea JMP ET1 are semnificația „salt la eticheta ET1”. Când programul este translatat în cod maşină, compilatorul (asamblorul) generează o valoare binară concretă pentru eticheta ET1, valoare care va fi integrată în codul instrucțiunii şi care va avea semnificația de adresă de salt (JMP *adr*). Salturile directe se implementează în variantă relativă şi nu absolută, în sensul că în codul instrucțiunii se codifică de fapt un OFFSET şi nu o adresă de salt absolută (vezi explicațiile relative la instrucțiunile de salt aferente procesorului didactic, paragraful 2.7.1.I, figurile 2.42 şi 2.43, precum şi 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 R_5”.

- **salturi indexate**. Sunt salturile în care adresa de salt se obține prin însumarea a două componenete: registrul specificat + indexul specificat. De exemplu, instrucțiunea JMP 18(R3) are semnificația „salt la adresa R_3+18”.

Prin urmare, instrucțiunile de salt sunt instrucțiuni viabile cu trei moduri de adresare: imediat, indirect şi 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 şi respectiv cod ilegal cu AD), cu precizarea că **operandul instrucțiunii va fi interpretat ca adresă de salt**.

Salturile directe revendică codificarea *offset*-ului şi 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 şi 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 şi *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 şi nu *offset*. Codul unei asemenea instrucțiuni de salt se întinde tot pe 32 de biți (două cuvinte succesive de memorie) şi este redat în figura 4.6.b).

Salturile indirecte nu utilizează constante; pentru generarea adresei de salt acestea utilizează doar un registru general. Prin urmare, codul unei asemenea instrucțiuni rămâne pe 16 biți şi 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 şi OFFSET-ul. OFFSET-ul este un număr cu semn (reprezentat în cod complementar) care indică sensul saltului şi 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_curent 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 şi prin urmare, programul va continua secvențial.

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

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

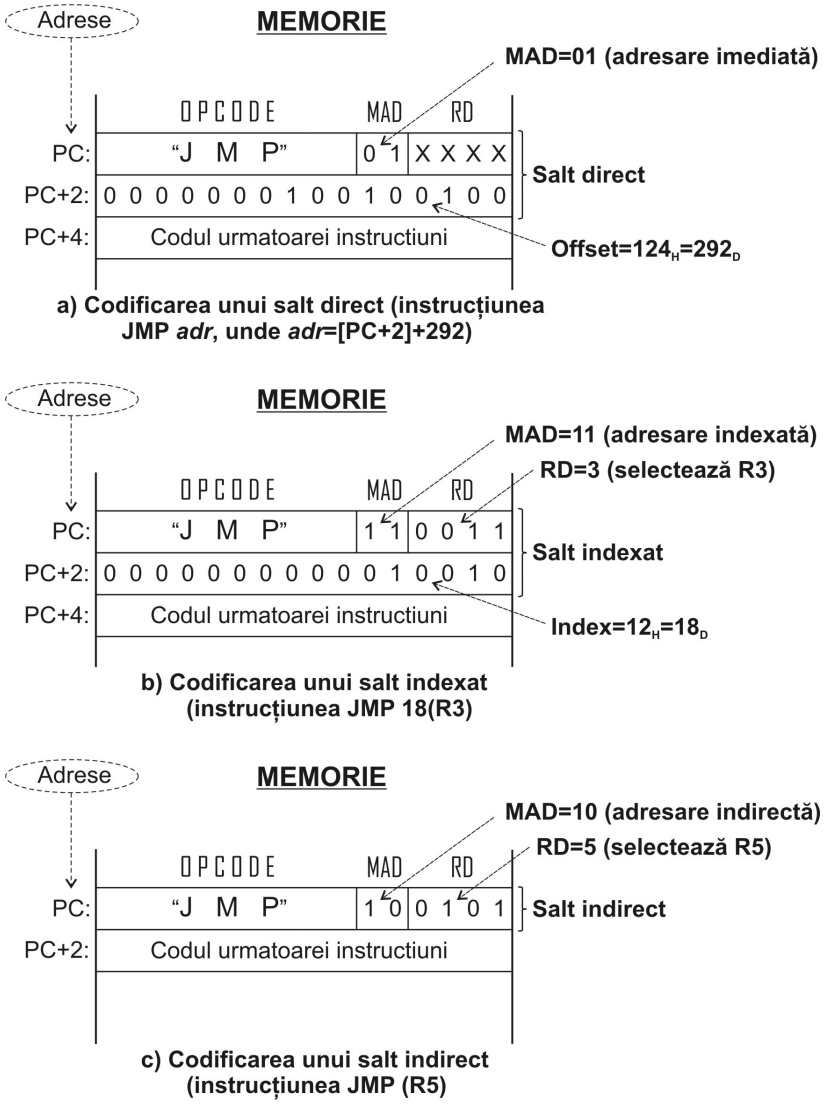


Fig 4.6 Formatul instrucțiunilor salt

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 *adr* dacă *flag*-ul Z=0 (*branch if not equal*)

BMI *adr* ; salt la adresa *adr* dacă *flag*-ul S=1 (*branch if minus*)

BPL *adr* ; salt la adresa *adr* dacă *flag*-ul S=0 (*branch if plus*)

BCS *adr* ; salt la adresa *adr* dacă *flag*-ul C=1 (*branch if carry is set*)

BCC *adr* ; salt la adresa *adr* dacă *flag*-ul C=0 (*branch if carry is clear*)

BVS *adr* ; salt la adresa *adr* dacă *flag*-ul V=1 (*branch is overflow is set*)

BVC *adr* ; salt la adresa *adr* dacă *flag*-ul V=0 (*branch if overflow is clear*)

- 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 R_7+24

BEQ (R1) ; dacă Z=1: salt la adresa specificată de R_1
; dacă Z=0: treci secvențial la următoarea instrucțiune

CALL ET1 ; apelul procedurii de la eticheta ET1.

Observație:
În ideea de a uşura 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:

OFFSET=Adresa_Proc1-PC_revenire (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 şi care, pe cale de consecință, nu pot fi încadrate în clasele anterioare. Mai exact, clasa conține instrucțiuni fără operanzi şi instrucțiuni cu operanzi referiți implicit (prin OPCODE) şi nu explicit prin câmpuri de adresă registru (RS, RD) şi respectiv mod de adresare (MAS, MAD) codificate în formatul intrucțiunii. Concret, clasa va conține: instrucțiuni de control, instrucțiuni de poziționare a *flag*-urilor de condiții, instrucțiuni de revenire din procedură şi respectiv din *handler*-ul de întrerupere, etc.

Formatul instrucțiunilor fără operanzi este redat în figura 4.7.

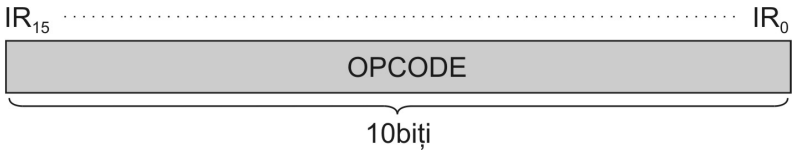


Fig 4.7 Formatul instrucțiunilor fără operanzi

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

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

NOP	; nici o operaţie (*No OPeration*)
HALT	; oprire
EI	; validare întreruperi (*Enable Interrupts*)
DI	; invalidare întreruperi (*Disable Interrupts*)
PUSH PC	; stiva←PC
POP PC	; PC←conţinutul locaţiei din vârful stivei
PUSH FLAG	; stiva←FLAG (registrul de *flag*-uri)
POP FLAG	; FLAG←conţinutul locaţiei din vârful stivei
RET	; revenire din procedură (*RETurn*)
IRET	; revenire din întrerupere (*Interrupt RETurn*)

4.1.3 Modurile de adresare

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

Câmpurile MAS (biţii IR_{11} şi IR_{10}) şi MAD (biţii IR_5 şi IR_4) sunt câmpuri de doi biţi, care permit codificarea a patru moduri de adresare (tabelele 4.2 şi 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 şi respectiv destinație), atunci operandul sursă se va afla în registrul general selectat de adresa codificată binar în câmpul RS iar operandul destinație se va afla în registrul general selectat de adresa codificată binar în câmpul RD. În mod uzual, registrul general selectat de câmpul RS se numeşte **registru sursă** iar registrul selectat de câmpul RD se numeşte **registru destinație**. Prin urmare, în cazul modului de adresare direct (atât la sursă cât şi la destinație), operandul sursă se află în registrul sursă iar operandul destinație se află în registrul destinație.

Fie instrucțiunea de adunare:

ADD R2,R8 ; $R_2 \leftarrow R_2+R_8$

Această instrucțiune adună registrul R_2 (operandul destinație) cu registrul R_8 (operandul sursă) şi depune rezultatul în registrul R_2. Prin urmare, instrucțiunea utilizează adresare directă atât la sursă cât şi la destinație. În figura 4.8 este evidențiat codul maşină al instrucțiunii precum şi 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 R_8 (conținutul registrului R_8)

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

RD=0010_B=2_D -operandul destinație este R_2 (conținutul registrului R_2)

Observaţie:
Unitatea adresabilă în memorie este octetul, iar codul instrucţiunii este pe 16 biţi (2 octeţi). Dacă instrucţiunea ADD R2,R8 *se află în memorie la adresa (generică) PC, atunci următoarea instrucţiune se va afla la adresa PC+2.*

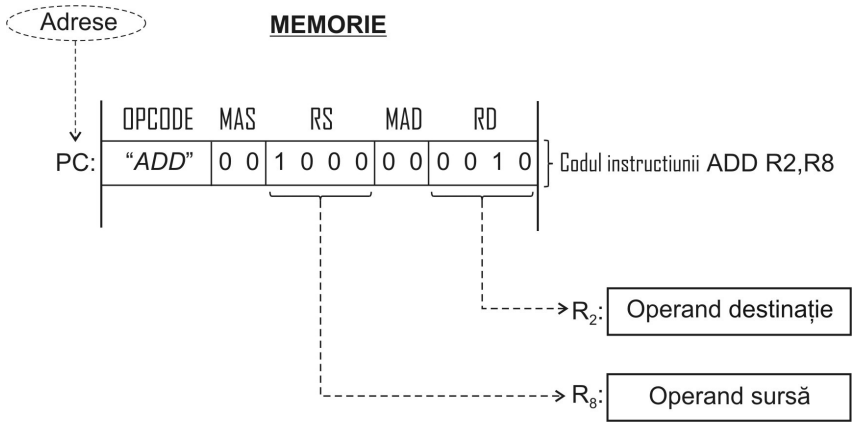


Fig 4.8 Localizarea operanzilor sursă şi destinaţie în cazul adresării directe

B. Modul de adresare imediat

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

Fie instrucţiunea:

MOV R0,124H ; $R_0 \leftarrow 124_H$ („*move*")

Operandul sursă al acestei instrucţiuni este constanta $124_H=0000000100100100_2$ (adresare imediată) iar operandul destinaţie este registrul R_0 (adresare directă). Operandul imediat va fi stocat în memorie în imediata vecinătate a codului instrucţiunii (fig. 4.9). Dacă instrucţiunea se află în memorie, la adresa (generică) PC, atunci operandul imediat va fi memorat la adresa PC+2 (în următoarea locaţie de memorie) iar codul următoarei instrucţiuni (din cadrul programului) va fi memorat la adresa PC+4. În realitate, codul instrucţiunii MOV R0,124H înglobează şi operandul imediat, va ocupa două cuvinte succesive în memorie şi va avea deci lungime dublă (2×16=32 de biţi). Iată cum la CISC se ajunge la formatul de lungime variabilă; lungimea variabilă a instrucţiunii CISC nu este un deziderat al proiectantului ci o consecinţă inevitabilă şi nedorită.

În final, să evidenţiem valorile câmpurilor din codul (din formatul) instrucţiunii exemplificate:

OPCODE="*MOV*" -o combinaţie binară care reprezintă *opcode*-ul instrucţiunii MOV

MAS=01 -adresare imediată la sursă (tabelul 4.2)

RS=XXXX_B -în cazul adresării imediate, câmpul RS nu este utilizat

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

RD=0000_B=0_D -operandul destinație este R0 (conținutul registrului R0)

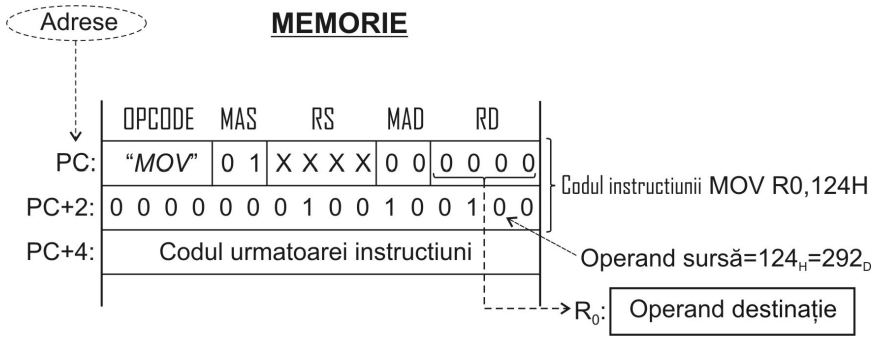


Fig 4.9 Localizarea operandului sursă în cazul modului de adresare imediat

C. Modul de adresare indirect

Modul de adresare indirect este utilizat pentru adresarea operanzilor 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 Rk se utilizează pentru adresarea directă. Conținutul registrului R_k reprezintă chiar operandul.

- Notația (Rk) 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) ; $MEM|_{R7} \leftarrow MEM|_{R7} + MEM|_{R9}$

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

rezultatul în locația de memorie adresată de R_7. Prin urmare, instrucțiunea utilizează adresare indirectă, atât la sursă cât şi la destinație. Codul maşină al instrucțiunii precum şi schema care evidențiază modul de localizare a celor doi operanzi sunt prezentate în figura 4.10.

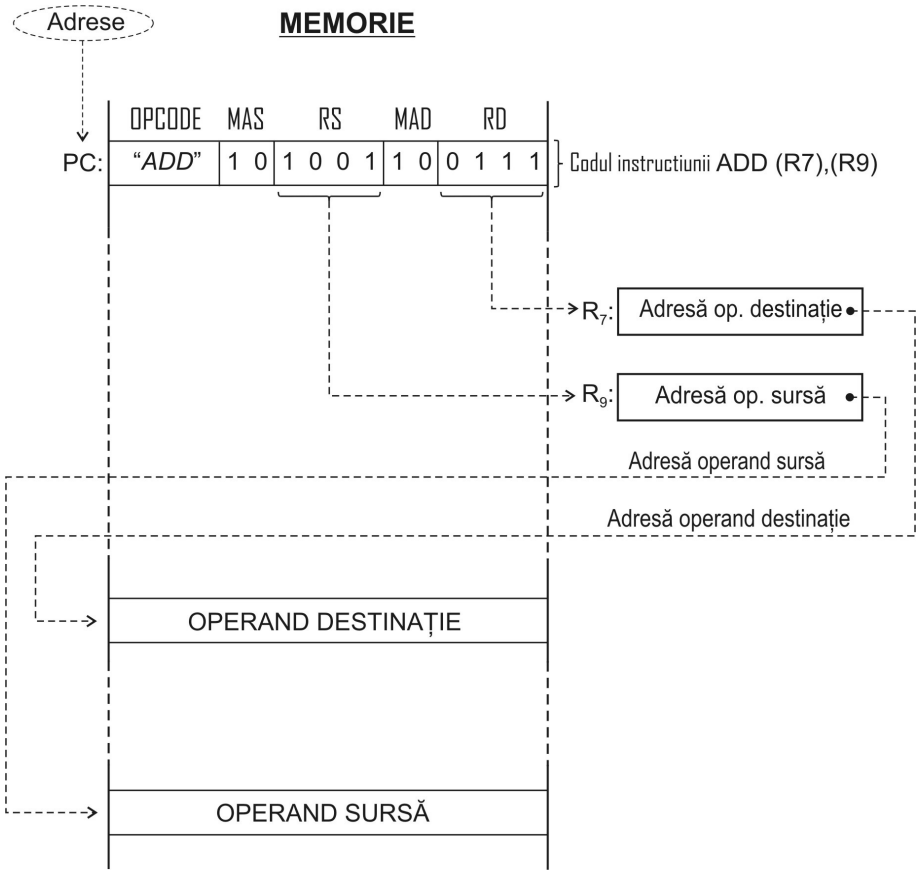


Fig 4.10 Localizarea operanzilor sursă şi destinație în cazul adresării indirecte

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 destinție (tabelul 4.3)

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

D. Modul de adresare indexat

Modul de adresare indexat, ca şi modul indirect, este utilizat pentru adresarea operanzilor aflaţi în memorie. După cum am vazut deja, la adresarea indirectă, adresa operandului se obţine din registrul specificat (indirectare prin registru). La adresarea indexată, adresa operandului sursă (destinaţie) se obţine pritr-o operaţie de adunare. Se adună la registrul sursă (registrul destinaţie) indexul sursă (indexul destinaţie) codificat binar pe 16 biţi şi plasat în memorie imediat după codul instrucţiunii. Dacă instrucţiunea are adresare indexată atât la sursă cât şi la destinaţie (ca în figura 4.11), atunci indexul sursă va fi plasat imediat după codul instrucţiunii (la adresa PC+2) iar indexul destinaţie va fi plasat la adresa PC+4. La adresa PC+6 se va găsi codul următoarei instrucţiuni. Dacă instrucţiunea are adresare indexată doar la unul dintre operanzi (fie sursă, fie destinaţie), atunci indexul (sursă sau destinaţie) va fi plasat imediat după codul instrucţiunii (la adresa PC+2); la adresa PC+4 va fi codul următoarei instrucţiuni.

Fie instrucţiunea:

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

Această instrucţiune are adresare indexată, atât la sursă cât şi la destinaţie. În figura 4.11 este prezentat codul maşină şi schema de localizare a operanzilor (sursă şi respectiv destinaţie) aferenţi acestei instrucţiuni. Valorile câmpurilor din codul (formatul) instrucţiuni exemplificate sunt:

OPCODE="*MOV*" -o combinaţie binară care reprezintă *opcode*-ul instrucţiunii MOV)

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

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

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

RD=0011_B=3_D -registrul R_3 furnizează o adresă de bază pentru operandul destinaţie

Operandul sursă al acestei instrucţiuni este reprezentat de conţinutul locaţiei 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=0000000000011000_2 este indexul sursă. Adresa operandului sursă se obţine prin însumarea celor două componente: adresa de bază (preluată din registrul sursă) + index sursă.

Similar, operandul destinaţie aferent acestei instrucţiuni este reprezentat de conţinutul locaţiei de memorie de la adresa R_3+142_D; R_3 este registrul destinaţie (care furnizează o adresă de bază relativă la operandul destinaţie) iar 142_D=0000000010001110_2 este indexul destinaţie. Adresa operandului se obţine prin însumarea celor două componente: adresa de bază (preluată din registrul destinaţie) + index destinaţie.

Din figura 4.11 se poate observa că instrucţiunea MOV 142(R3),24(R8) are codul maşină extins la 3×16=48 biţi. Pe primii 16 biţii s-a codificat codul propriu-zis al instrucţiunii, pe următorii 16 biţii s-a codificat indexul sursă iar pe ultimii 16 biţi s-a codificat indexul destinaţie. Putem în sfârşit 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 şi pe 48 biți.

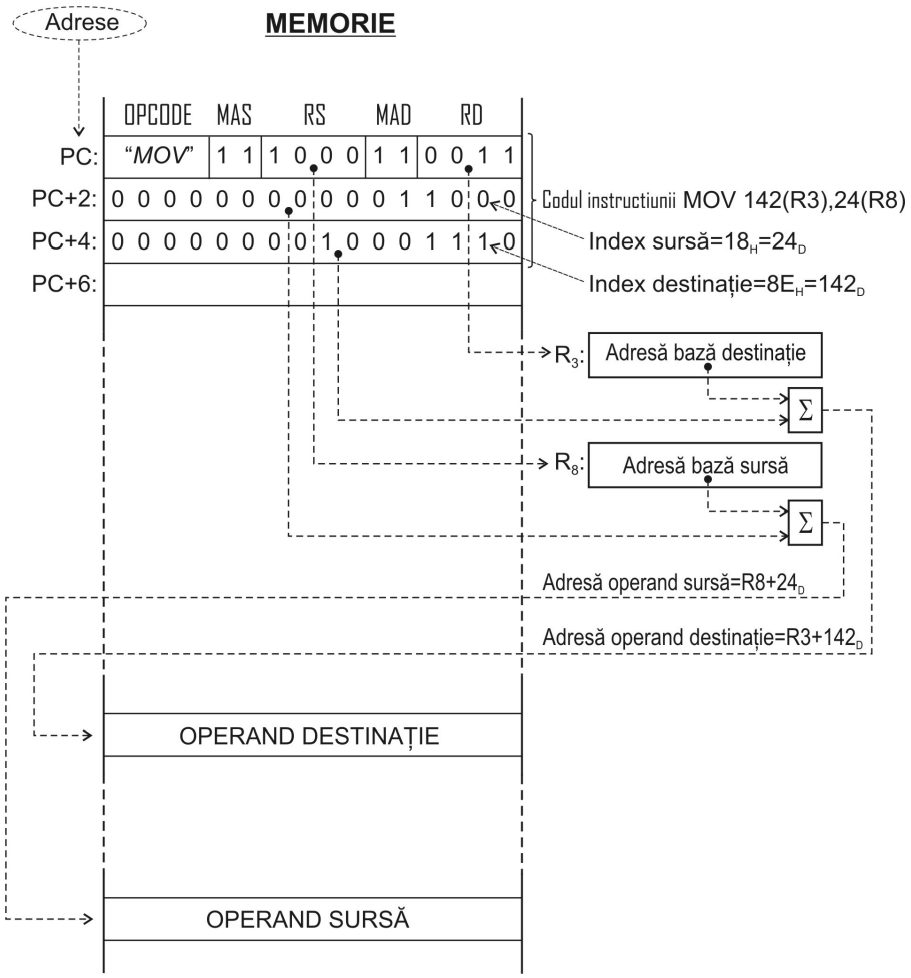


Fig 4.11 Localizarea operanzilor sursă şi destinație în cazul adresării indexate

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 deja: adresarea imediată la destinație reprezintă cod ilegal.

În figura 4.12 este prezentat codul maşină şi 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 şi din figura 4.12, operandul sursă este constanta $26F_H=623_D$ iar operandul destinație este conținutul locației de memorie selectată de adresa R_7+62_D. Această instrucțiune „*compare*" compară operandul sursă (*src*) cu operandul destinație

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

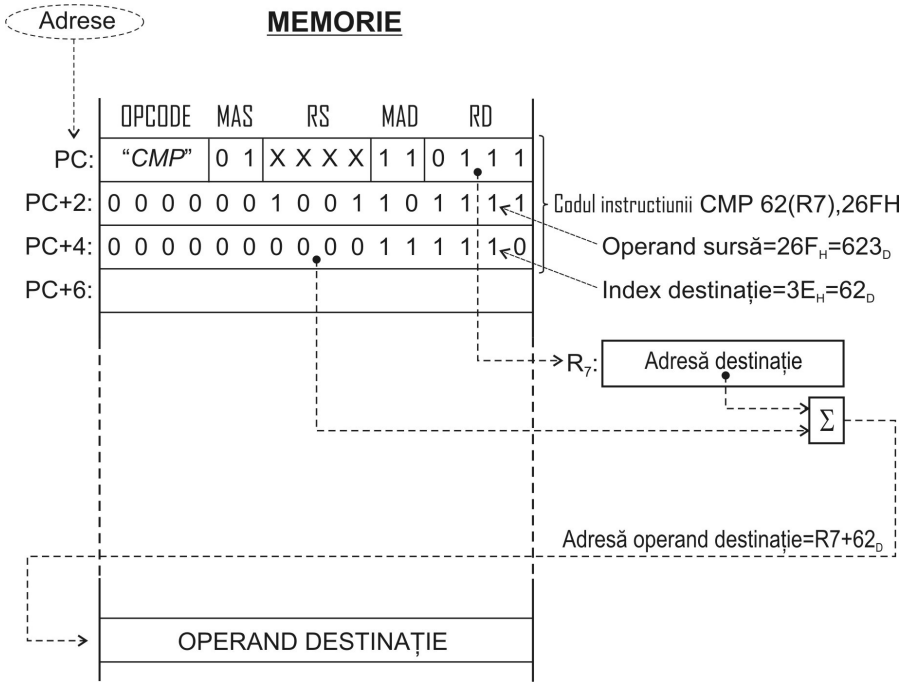


Fig 4.12 Localizarea operanzilor sursă şi destinație aferenți instrucțiunii CMP 62(R7),26FH

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 şi 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 şi 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 şi 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 (imediat, direct, indirect şi indexat) şi respectiv 3 moduri de adresare la operandul destinație (adresarea imediată la destinație reprezintă cod ilegal). Cele 4 moduri de adresare la sursă şi 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 şi MAD din codul maşină al instrucțiunii (figura 4.4).

O instrucțiune generică din clasele B şi C generează doar 3 instrucțiuni (combinații) 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 şi respectiv AD ilegal la clasa C).

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

Prin urmare, câmpul OPCODE trebuie să asigure:

- codificarea celor 4 clase (A, B, C şi 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.

IR_{15}	IR_{14}	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 (IR_{15}÷IR_{12}) iar biții IR_{15} şi IR_{14} au fost deja alocați pentru codificarea clasei. Rezultă că rămân doar 2 biți (IR_{13} şi IR_{12}) 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 şi D, unde avem câmpuri OPCODE mult mai extinse. Prin urmare, va trebui să revenim la codificarea claselor şi să aplicăm un artificiu prin care, pentru clasa A, să utilizăm un singur bit (IR_{15}) pentru codificarea clasei. Ne vor rămâne astfel 3 biți (IR_{14}, IR_{13} şi IR_{12}) pentru a codifica cele 7 instrucțiuni din cadrul acestei clase. Artificiul pe care îl propunem este redat în tabelul 4.5.

IR_{15}	IR_{14}	IR_{13}	CL_1	CL_0	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 –defineşte clasa A

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

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

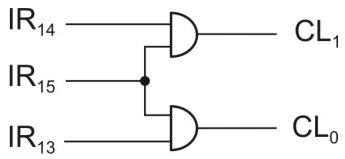


Fig 4.13 Generarea semnalelor care codifică cele 4 clase de instrucțiuni

Pe baza celor stabilite se poate trece la codificarea intregului set de instrucțiuni. Modul de codificare a celor 4 clase şi respectiv a instrucțiunilor din cadrul fiecărei 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 defineşte clasa A iar următorii 3 biți din OPCODE (IR_{14}, IR_{13} şi IR_{12}) codifică cele 7 instrucțiuni din cadrul clasei A.

- IR_{15}, IR_{14}, IR_{13}=1,0,1 defineşte clasa B iar biții IR_{11}, IR_{10}, IR_9 şi IR_8 codifică cele 13 instrucțiuni din cadrul clasei B.

- IR_{15}, IR_{14}, IR_{13}=1,1,0 defineşte clasa C iar biții IR_{11}, IR_{10}, IR_9 şi IR_8 codifică cele 10 instrucțiuni din cadrul clasei C.

- IR_{15}, IR_{14}, IR_{13}=1,1,1 defineşte clasa D iar biții IR_{11}, IR_{10}, IR_9 şi 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	INSTRUCTIUNE	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	MAD		RD			
	NEG	1	0	1	0	0	0	0	1	0	0						
	INC	1	0	1	0	0	0	1	0	0	0						
	DEC	1	0	1	0	0	0	1	1	0	0						
	ASL	1	0	1	0	0	1	0	0	0	0						
	ASR	1	0	1	0	0	1	0	1	0	0						
	LSR	1	0	1	0	0	1	1	0	0	0						
	ROL	1	0	1	0	0	1	1	1	0	0						
	ROR	1	0	1	0	1	0	0	0	0	0						
	RLC	1	0	1	0	1	0	0	1	0	0						
	RRC	1	0	1	0	1	0	1	0	0	0						
	PUSH Ri	1	0	1	0	1	0	1	1	0	0						
	POP Ri	1	0	1	0	1	1	0	0	0	0						
C	BEQ	1	1	0	0	0	0	0	0	0	0	MAD		RD			
	BNE	1	1	0	0	0	0	0	1	0	0						
	BMI	1	1	0	0	0	0	1	0	0	0						
	BPL	1	1	0	0	0	0	1	1	0	0						
	BCS	1	1	0	0	0	1	0	0	0	0						
	BCC	1	1	0	0	0	1	0	1	0	0						
	BVS	1	1	0	0	0	1	1	0	0	0						
	BVC	1	1	0	0	0	1	1	1	0	0						
	JMP	1	1	0	0	1	0	0	0	0	0						
	CALL	1	1	0	0	1	0	0	1	0	0						
D	CLC	1	1	1	0	0	0	0	0	1	1	1	1	0	1	1	1
	CLZ	1	1	1	0	0	0	0	0	1	1	1	1	1	0	1	1
	CLS	1	1	1	0	0	0	0	0	1	1	1	1	1	1	0	1
	CLV	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	0
	CCC	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0
	SEC	1	1	1	0	0	0	0	1	0	0	0	0	1	0	0	0
	SEZ	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	0
	SES	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1	0
	SEV	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	1
	SCC	1	1	1	0	0	0	0	1	0	0	0	0	1	1	1	1
	NOP	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
	HALT	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0
	EI	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
	DI	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0
	PUSH PC	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0
	POP PC	1	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0
	PUSH FLAG	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
	POP FLAG	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0
	RET	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
	IRET	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0

Fig 4.14 Codificarea instrucțiunilor

***Observaţie*:**
Să remarcăm că primele 5 instrucţiuni din clasa D (instrucţiunile de resetare a flag-urilor de condiţii) sunt codificate cu un OPCODE comun pe zona $IR_{15} \div IR_8$ (pe octetul cel mai semnificativ din codul instrucţiunii). Diferenţierea acestor instrucţiuni este realizată de octetul inferior ($IR_7 \div IR_0$) unde am codificat de fapt o mască. În microrutina de execuţie aferentă acestor instrucţiuni vom realiza operaţia (4.2).

$$FLAG \leftarrow FLAG \text{ AND } IR_7 \div IR_0 \qquad (4.2)$$

Operandul $IR_7 \div IR_0$ reprezintă deci masca, care va fi emisă pe DBUS şi aplicată pe intrarea ALU cu extensie de semn pe octetul superior (figura 4.15). Valoarea măştii a fost stabilită ţinând cont de poziţia celor 4 flag-uri (4 bistabili) de condiţii în cadrul registrului FLAG (figura 4.16).

Prin extensia de semn valoarea măştii nu se modifică. Prin urmare, se va face o operaţie AND între conţinutul registrului FLAG (operandul 1) şi masca emisă cu extensie de semn (operandul 2) şi rezultatul va fi depus în registrul FLAG. Biţii care au valoarea 0 în mască vor determina resetarea biţilor corespondenţi din registrul FLAG. Biţii care au valoarea 1 în mască vor lăsa nemodificaţi biţii corespondenţi din registrul FLAG. Prin urmare, microrutina de execuţie aferentă acestor 5 instrucţiuni va fi una comună (vezieticheta CLC: din microprogramul de emulare - tabelul 4.9), în care se va executa operaţia (4.2); flag-ul ce va fi resetat va fi decis exclusiv de mască.

O strategie similară vom aplica şi pentru următoarele 5 instrucţiuni din clasa D (instrucţiunile de setare a flag-urilor de condiţii). În mod analog, aceste 5 instrucţiuni vor avea o microrutină de execuţie comună în care se va efectua operaţia (4.3).

$$FLAG \leftarrow FLAG \text{ OR } IR_7 \div IR_0 \qquad (4.3)$$

Se va face deci o operaţie OR între conţinutul registrului FLAG (operandul 1) şi masca codificată în octetul inferior al instrucţiunii (care va fi emisă cu extensie de semn pe octetul superior) şi rezultatul va fi depus în registrul FLAG. Biţii care au valoarea 1 în mască vor determina setarea biţilor corespondenţi din registrul FLAG. Biţii care au valoarea 0 în mască vor lăsa nemodificaţi biţii corespondenţi din registrul FLAG. Prin urmare, microrutina de execuţie aferentă acestor 5 instrucţiuni va putea fi una comună (vezieticheta SEC: din microprogramul de emulare - tabelul 4.9) în care se va efectua operaţia (4.3); flag-ul ce va fi setat va fi decis exclusiv de mască.

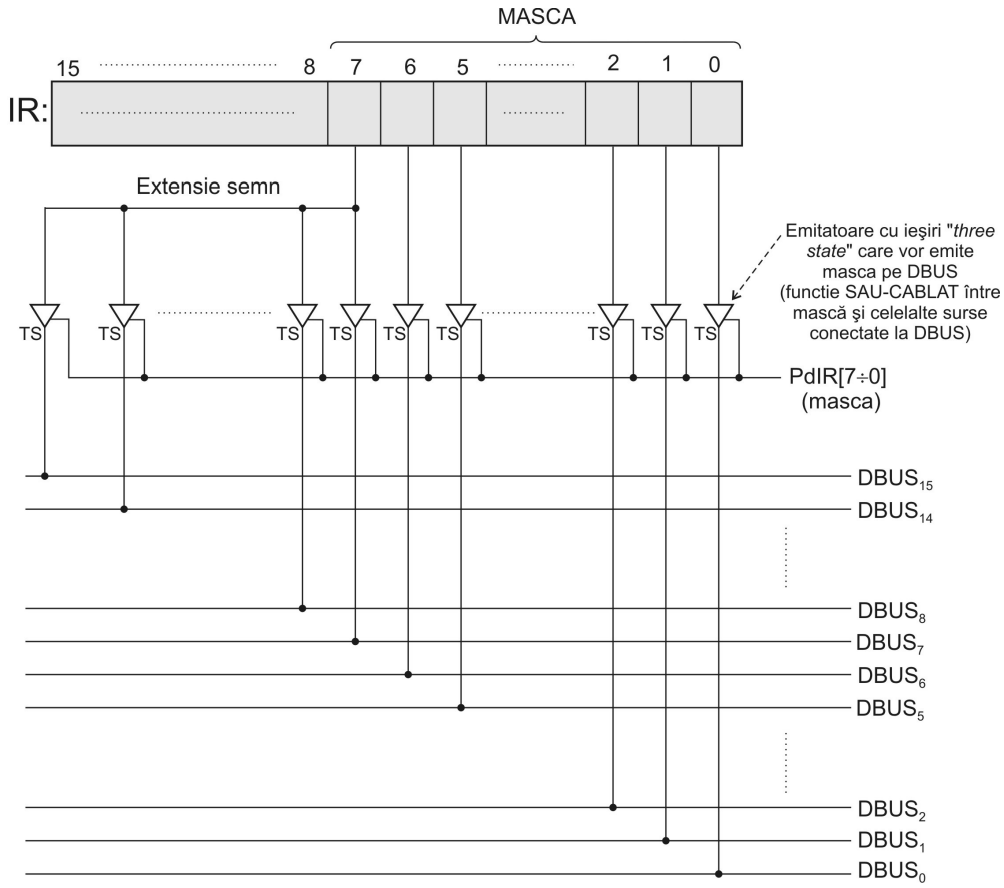


Fig 4.15 Emisia măştii pe DBUS cu extensie de semn pe octetul superior

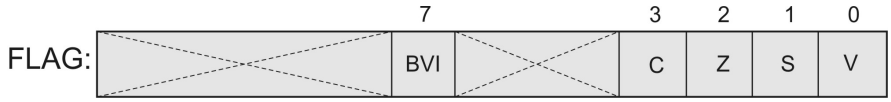


Fig 4.16 Registrul FLAG

Să mai observăm că, pentru clasele B, C şi D am utilizat aceeaşi zonă de OPCODE (zona IR_{11}÷IR_8) pentru codificarea instrucţiunilor (în cadrul clasei). În cadrul microprogramului de emulare, microrutinele de execuţie vor fi selectate printr-un salt indexat iar indexul utilizat va fi tocmai câmpul IR_{11}÷IR_8. Rezultă că, pentru cele 3 clase (B, C şi D), se va utiliza acelaşi index pentru selecţia microrutinelor de execuţie. Reducerea numărului de indecşi conduce la simplificarea *harware*-ului care implementează blodul de selecţie index.

Observaţie:
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 cere ţinteşte saltul indexat respectiv. Dacă microrutinele au lungimea de 2 microinstructiuni atunci, vom concatena un singur bit de zero în dreapta indexului şi salturile se vor executa din 2 în 2 microadrese (microinstrucţiuni). Dacă microrutinele au lungimea de 4 microinstructiuni atunci, vom concatena 2 biţi de zero în dreapta indexului şi salturile se vor executa din 4 în 4 microadrese (vezi 3.4.1.B). Dacă microrutinele au lungimea de o singură microinstrucţiune, atunci nu vom concatena zerouri şi prin urmare salturile se vor face pe microadrese (microinstrucţiuni) succesive.
Pe baza evaluării lungimii microrutinelor de execuţie am ajuns la următoarele concluzii:

- *Pentru clasa* B*, cele mai multe microrutine de execuţie au lungimea de o singură microinstrucţiune; pentru selecţia acestora vom utiliza* $INDEX_5=IR_{11}\div IR_8$ *(fără zerouri concatenate la dreapta; vezi figura 4.17).*

- *Pentru clasele* C *şi* D*, cele mai multe microrutine de execuţie au lungimea de două microinstrucţiuni; pentru selecţia 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 instrucţiunilor (în cadrul clasei) este realizată cu alţi 3 biţi din OPCODE *(biţii* $IR_{14}\div IR_{12}$*). Pe de altă parte, cele mai multe microrutine de execuţie din clasa A au lungimea de o singură microinstrucţiune. Prin urmare, pentru selecţia 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 indecşilor va putea fi stabilită doar după proiectarea microrutinelor din cadrul microprogramului de emulare şi după evaluarea lungimii acestora. Doar după această evaluare de lungime se poate decide numărul de zerouri concatenate la dreapta fiecărui index.

În final să menţinăm că, pentru codificarea instrucţiunilor există multiple soluţii iar proiectantul trebuie să urmeze strategii de optimizare şi la acest nivel. Soluţia pe care noi am adoptat-o (sintetizată în figura 4.14) vizează simplificarea decodificatorului instrucţiunii şi reducerea numărul de indecşi necesari pentru selecţia microrutinelor de execuţie. Reducerea numărului de indecşi conduce la simplificarea blocului de selecţie index.

4.2 Unitatea de control microprogramată

În cele ce urmează vom parcurge etapele de proiectare a unităţii de control microprogramate. Soluţiile pe care le vom adopta pe parcursul diverselor faze de proiectare vor viza maximizarea performanţelor procesorului CISC definit la 4.1.

4.2.1 Organizarea microprogramului de emulare

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

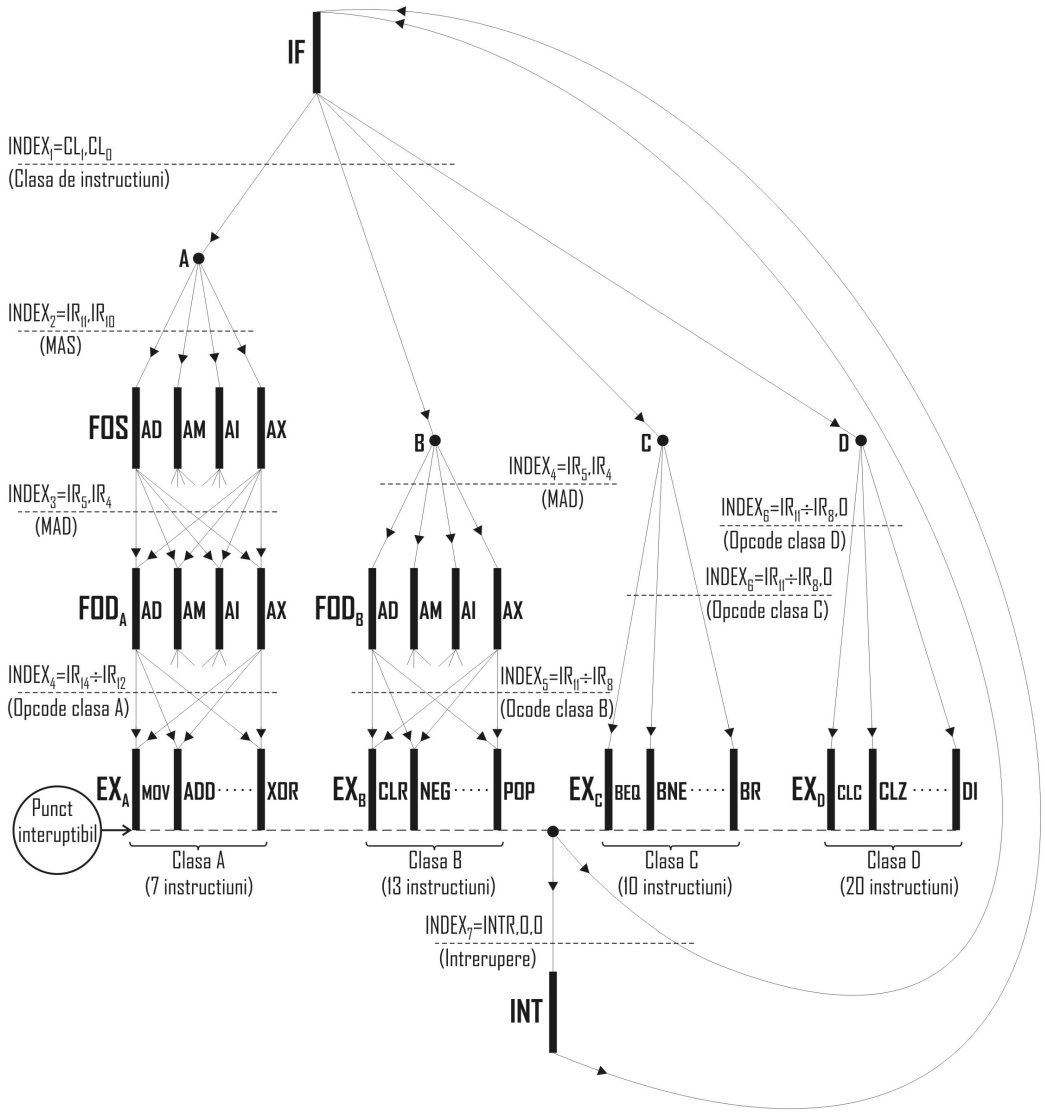


Fig 4.17 Organizarea microprogramului de emulare

Procesarea unei instrucțiuni (emularea instrucțiunii în microcod) începe cu microrutina ***Instrucțion Fetch*** (**IF**), prin care instrucțiunea este citită din memorie şi 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 şi generarea semnalelor CL_1 şi CL_0 (figura 4.13) care vor preciza clasa din care face

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

La ieşirea 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 $INDEX_1=CL_1, CL_0$.

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**) şi de o microrutină de **Execuție** (**EX_A**). La ieşirea din microrutina de execuție se implementează punctul interuptibil (se testează cererea globală de întrerupere INTR). Pentru creşterea vitezei de procesare, punctul interuptibil va fi implementatt tot printr-un salt indexat, cu indexul $INDEX_7$=INTR, 0, 0.

  Există 4 microrutine **FOS** şi respectiv 4 microrutine **FOD_A** (cu precizarea că **FOD_A** cu adresare imediată reprezintă de fapt cod ilegal). Cu **FOD_A** am notat colecția de 4 microrutine ***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 $INDEX_2=IR_{11}, IR_{10}$.

  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 $INDEX_3=IR_5, IR_4$.

  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 $INDEX_4=IR_{14}\div IR_{12}$.

  Să notăm că, microrutina **FOS** selectată, va localiza operandul sursă (conform cu modul de adresare codificat în câmpul MAS), îl va citi şi-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 şi-l va încărca în procesor (în registrul tampon, MDR). Microrutina **EX_A** selectată, va efectua operația (specificată prin OPCODE) asupra celor 2 operanzi şi 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 ieşirea din microrutina de execuție se implementează punctul interuptibil (identic cu cel implementat la clasa A: salt indexat, cu indexul $INDEX_7$=INTR, 0, 0). Selecția microrutinei **FOD_B** (una din 4) se va realiza tot printr-un salt indexat, cu indexul $INDEX_3=IR_5, IR_4$ (exact ca la

clasa A). Selecția microrutinei **EX_B** (una din 13) se va realiza tot printr-un salt indexat, cu indexul $INDEX_5=IR_{11}\div IR_8$ (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** şi **FOD_B**) pentru a simplifica intrarea în microrutinele **EX_A** şi respectiv **EX_B**. Dacă am fi utilizat acelaşi set de microrutine **FOD** pentru ambele clase de instrucțiuni atunci, la ieşirea din microrutinele **FOD,** ar fi trebuit implementate două mecanisme de ramificație succesive: primul în două direcții (clasa A şi respectiv B) şi al doilea pe microrutina aferentă instrucțiunii din cadrul clasei. O astfel de ramificație 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 microprogramul de emulare, va fi nesemnificativ.

- Dacă instrucțiunea este din clasa C (instrucțiuni de salt), urmează direct microrutina de **Execuție** (**EX_C**). La ieşirea din microrutina de execuție se implementează punctul interuptibil (identic cu cel implementat la clasele A şi B: salt indexat, cu indexul $INDEX_7$=INTR, 0, 0).

  Selecția microrutinei **EX_C** (una din 10) se va realiza tot printr-un salt indexat, cu indexul $INDEX_5=IR_{11}\div IR_8$, 0 (OPCODE-ul propriu-zis pentru clasa C, urmat de un bit de zero deoarece am rezervat o lungime medie de 2 microinstrucțiuni pentru microrutinele **EX_C**).

- Dacă instrucțiunea este din clasa D (instrucțiuni fără operanzi), urmează direct microrutina de **Execuție** (**EX_D**). La ieşirea din microrutina de execuție se implementează punctul interuptibil (identic cu cel implementat la clasele A, B şi C: salt indexat, cu indexul $INDEX_7$=INTR, 0, 0).

  Selecția microrutinei **EX_D** (una din 20) se va realiza tot printr-un salt indexat, cu indexul $INDEX_5=IR_{11}\div IR_8$, 0 (OPCODE-ul propriu-zis pentru clasa C, urmat de un bit de zero deoarece am rezervat o lungime medie de 2 microinstrucțiuni pentru microrutinele **EX_D**).

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

Dacă în punctul interuptibil se detectează INTR=0 atunci, după microrutina de execuție, urmează microrutina ***Fetch Instrucțiune*** (se va face *fetch*-ul următoarei instrucțiuni din cadrul programului). Dacă în punctul interuptibil se detectează INTR=1 atunci microrutina de execuție este urmată de microrutina de întrerupere (INT) care are sarcina de a salva în stivă registrele FLAG şi PC şi 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 şi permit ramificații complexe în microcod)

Pentru proiectarea microinstrucțiunii generale vom urma paşii 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 şi 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, ieşirile 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 genelală 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 şi DESTINAȚIE RBUS provin din microinstrucțiunea de transfer sincron. Cu ajutorul comenzilor codificate în aceste câmpuri se controlează transferurile de date registru-registru (transferurile interne). Informația transferată traversează întotdeauna unitatea ALU şi 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 şi rotire le-am codificat în câmpul OPERAȚIE ALU. Prin urmare, operațiile de deplasare şi rotire le vom implementa în ALU (vezi cele 2 observații de la 2.7.2.D).

35 ····· 32	31 ····· 28	27 ····· 24	23 ····· 20	19 18	17 ····· 14	13 ····· 11	10 ····· 8	7	6 ····· 0
SURSA SBUS	**SURSA DBUS**	**OPERATIE ALU**	**DESTINATIE RBUS**	**OPERATII CU MEMORIA**	**ALTE OPERATII**	**SUCCESOR**	**SELECTIE INDEX**	**$\overline{T}$/F**	**MICROADRESA DE SALT**
NONE	NONE	NONE	NONE	NONE	NONE	STEP	$INDEX_0$	T	
$PdFLAG_S$	$PdFLAG_D$	SBUS	PmFLAG	IFCH	+2SP	JUMPI	$INDEX_1$	F	
$PdRG_S$	$PdRG_D$	DBUS	PmFLAG[3÷0]	RD	-2SP	IF ACLOW...	$INDEX_2$		
$PdSP_S$	$PdSP_D$	ADD	PmRG	WR	+2PC	IF CIL ...	$INDEX_3$		
PdT_S	PdT_D	SUB	PmSP		A(1)BE0	IF C...	$INDEX_4$		
$Pd\overline{T}_S$	$PdPC_D$	AND	PmT		A(1)BE1	IF Z...	$INDEX_5$		
$PdPC_S$	$PdIVR_D$	OR	PmPC		$PdCOND_A$	IF S...	$INDEX_6$		
$PdIVR_S$	$PdADR_D$	XOR	PmIVR		C_{IN},$PdCOND_A$	IF V...	$INDEX_7$		
$PdADR_S$	$Pd\overline{MDR}_D$	ASL	PmADR		$PdCOND_L$				
$PdMDR_S$	$Pd\overline{MDR}_D$	ASR	PmMDR		A(1)BVI				
$PdIR[7\div0]_S$	$PdIR[7\div0]_D$	LSR	-		A(0)BVI				
$Pd0_S$	$Pd0_D$	ROL	-		A(0)BPO				
$Pd\text{-}1_S$	$Pd\text{-}1_D$	ROR	-		INTA,-2SP				
-	-	RLC	-		A(0)BE,A(0)BI				
-	-	RRC	-		-				
-	-	-	-		-				

Fig 4.18 Microinstrucțiunea generală aferentă procesorului CISC

Câmpul OPERAŢII CU MEMORIA provine din microinstrucţiunea de transfer asincron. Prin urmare, în paralel cu un transfer sincron (registru-registru) se poate activa şi o operaţie de citire din sau de scriere în memorie. Dacă se activează o scriere (prin microcomanda WRITE), atunci va fi preluată data din MDR şi va fi scrisă în locaţia de memorie selectată de adresa existentă în ADR. Dacă se activează o citire prin microcomanda READ, atunci se vor citi informaţiile stocate în locaţia selectată de adresa prezentă în ADR şi informaţiile 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 informaţiile stocate în locaţia selectată de adresa prezentă în ADR şi informaţiile citite vor fi încărcate în registrul IR; astfel se realizează *fetch*-ul unei instrucţiuni din memorie.

Câmpul ALTE OPERAŢII a rezultat prin unificarea câmpurilor OTHER OPERATIONS (din microinstrucţiunea de transfer sincron) şi respectiv SHIFT & OTHER OPERATIONS (din microinstrucţiunea 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 extarge lista comenzilor din categoria „*other*" si toate comenzile din listă vor trebui codificate în câmpul ALTE OPERAŢII al microinstrucţiunii generale. Este momentul în care se poate deduce exact şi nmărul de biţi din componenţa acestui câmp.

În sfârşit, câmpurile SUCCESOR, SELECŢIE INDEX, $\overline{T}/F$ şi MICROADRESA DE SALT provin din microinstrucţiunea de ramificaţie. Numărul de biţi din componenţa 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 microinstrucţiuni (2^n locaţii în MPM) atunci câmpul MICROADRESA DE SALT va avea n biţi în componenţa sa. Câmpurile SUCCESOR şi SELECŢIE INDEX vor putea fi definitivate tot după proiectarea microcodului (microprogramului de emulare).

Pentru stabilirea succesorilor va trebui să avem lista tuturor condiţiilor testate în microcod. Înainte de proiectarea microcodului, putem doar intui condiţiile testate dar nu putem fi siguri că lista acestora este completă. Pentru procesorul nostru CISC, este clar că în microrutinele de execuţie aferente instrucţiunilor de *branch* (de salt condiţionat) vom testa cele 4 *flag*-uri de condiţii:C, Z, S şi V. Este clar de asemenea că vom testa condiţiile de excepţie ACLOW (căderea tensiunii de alimentare) şi CIL (cod ilegal). Pentru creşterea vitezei de operare a procesorului (vitezei de emulare în microcod a instrucţiunilor), punctul interuptibil îl vom implementa printr-un salt indexat cu $INDEX_7$=INTR, 0, 0 (vezi fig 4.17). În această situaţie, condiţia INTR nu va mai trebui testată şi prin urmare o vom elimina din listă.

Pornim deci de la premisa că lista de condiţii ce vor trebui testate este: ACLOW, CIL, C, Z, S, şi V. Având stabilită lista condiţiilor testate şi ţinând cont că bitul $MIR_7=\overline{T}/F$ specifică modul de test a condiţiei selectate (test pe fals sau pe adevărat), succesorii pot fi definiţi aşa cum indică tabelul 4.6.

O noutate faţă de succesorii definiţi la procesorul didactic (tabelul 3.2), ar fi apariţia succesorului STEP (explicit) în tabelul 4.6. Asta e, la procesorul didactic succesorul STEP a fost un succesor implicit al microinstrucţiunilor de transfer sincron şi asincron; el nu a fost codificat explicit în aceste microinstrucţiuni ci a fost dedus implicit de către SEQ, pornind de la tipul microinstrucţiunii (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 secvențială 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 si succesorii codificați în microinstrucțiune (va fi revizuit tabelul 4.6).

$MIR_{13\div11}$	Mnemonica succesorului	Funcția *f* testată
000	STEP	f=MIR_7; rezultă g=MIR_7 **XOR** MIR_7=0
001	JUMPI μADR+INDEX	f= $\overline{MIR_7}$; rezultă g= $\overline{MIR_7}$ **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 SELECȚIE INDEX are sarcina de a selecta indexul dorit. În figura 4.17 s-a stabilit structura microprogramului de emulare (microcodului) şi pe baza aceastei figuri se poate elabora lista inecşilor utilizați. Această listă este sintetizată în tabelul 4.7, unde este redată şi valoarea (binară) a fiecărui index. Valoarea a fost stabilită pornind de la indecşii definiți în figura 4.17. Dacă după proiectarea microcodului se va dovedi necesară redefinirea unuia sau mai multor indecşi, atunci se va reveni şi se va redefini şi tabelul 4.7.

Să observăm că primul index din tabelul 4.7 ($INDEX_0$) are valoarea 0. Cu ajutorul acestui index vom converti succesorii JUMPI din tabelul 4.6, în sucesori JUMP echivalenți (vezi explicațiile relative la tabelele 3.1 şi 3.2 de la 3.4.1). Succesori JUMP sunt prezentați 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 şi 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

$MIR_{10\div8}$	Denumire INDEX	Valoare INDEX (valoare binară)							Semnificaţie
		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 instructiuni
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 destinatie
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 si D
111	$INDEX_7$	0	0	0	0	INTR	0	0	Cererea globala de întrerupere

Tabelul 4.7 Indecşii definiţi

$MIR_{13\div11}$	Mnemonica succesorului	Funcţia *f* testată
000	STEP	f=MIR_7; rezultă g=MIR_7 **XOR** MIR_7=0
001	JUMP μADR	f= $\overline{MIR_7}$; rezultă g= $\overline{MIR_7}$ **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 microinstrucţiunii generale (pasul 4 = **transpunere**). La finele pasului 3 definit la 3.12 (figura 3.47 – varianta 1 de lucru) s-a obţinut microinstrucţiunea generală aferentă procesorului didactic. Pasul 4 presupune **transpunerea** microinstrucţiunii generale de pe o arhitectură pe alta (de pe arhitectura procesorului didactic pe arhitectura procesorului CISC). Această transpunere trebuie realizată pentru fiecare câmp al microinstrucţiunii generale obţinută la finele pasului 3. În figura 4.19, este exemplificată operaţia de transpunere pentru câmpul SURSA SBUS.

Procesorul Didactic

SURSA SBUS ($MIR_{17\div15}$)	Microcomenzi codificate
000	NONE
001	PdIR[OP]
010	PdIR[IND]
011	PdA
100	PdSP
101	Pd0$_S$
110	Pd-1$_S$
111	-

Transpunere SURSA SBUS →

Procesorul CISC

SURSA SBUS ($MIR_{35\div32}$)	Microcomenzi codificate
0000	NONE
0001	PdFLAG$_S$
0010	PdRG$_S$
0011	PdSP$_S$
0100	PdT$_S$
0101	Pd$\overline{T}_S$
0110	PdPC$_S$
0111	PdIVR$_S$
1000	PdADR$_S$
1001	PdMDR$_S$
1010	PdIR[7÷0]$_S$
1011	Pd0$_S$
1100	Pd-1$_S$
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 ($MIR_{17\div15}$) şi codifică microcomenzile care emit pe SBUS registrele IR[OP], IR[IND], A, SP şi constantele 0 şi -1 (vezi figurile 3.3 şi 2.2). La procesorul CISC avem alte registre conectate la SBUS (vezi figura 4.1). Prin urmare, câmpul SURSA SBUS trebuie să codifice comenzile aferente acestor registre (PdFLAG$_S$, PdRG$_S$, PdSP$_S$, PdT$_S$, Pd$\overline{T}_S$, PdPC$_S$, PdIVR$_S$, PdADR$_S$, PdMDR$_S$, PdIR[7÷0]$_S$) şi comenzi de predare a unor constante (Pd0$_S$ şi Pd-1$_S$). Am utilizat indicele „$_S$” deoarece, la procesorul CISC, comenzi similare vor fi codificate şi în câmpul SURSA DBUS; pentru acestea vom utiliza indicele „$_D$”. Pe baza acestor indici vom diferenția comenzile 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 ($MIR_{35\div32}$).

Constantele nu pot fi stabilite concret decât după proiectarea microprogramului de emulare. Nici comenzile de tipul Pd$\overline{T}_S$, care revendică şi 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 comenzile 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 şi controlul registrelor 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 registru pe SBUS
- citirea unui registru pe DBUS
- scrierea (încărcarea) într-un registru a datelor preluate de pe RBUS

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

- $PdRG_S$ –emite conținutul unui registru pe SBUS
- $PdRG_D$ –emite conținutul unui registru pe DBUS
- PmRG –scrie într-un registru 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 registrelor 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 dispună de două porturi de citire şi un port de scriere. În figura 4.20 este prezentată structura multiport care oferă aceste facilități.

Adresa $AC_{3A \div 0A}$ selectează registrul ce va fi citit pe portul de citire **A**, adresa $AC_{3B \div 0B}$ selectează registrul ce va fi citit pe portul de citire **B** iar adresa $AS_{3 \div 0}$ selectează registrul 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 şi respectiv RD din codul instrucțiunii curente; instrucțiunea curentă este instrucțiunea aflată în execuție (instrucțiunea al cărei cod este prezent în registrul IR).
Urmărind schema din figura 4.20 putem concluziona:

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

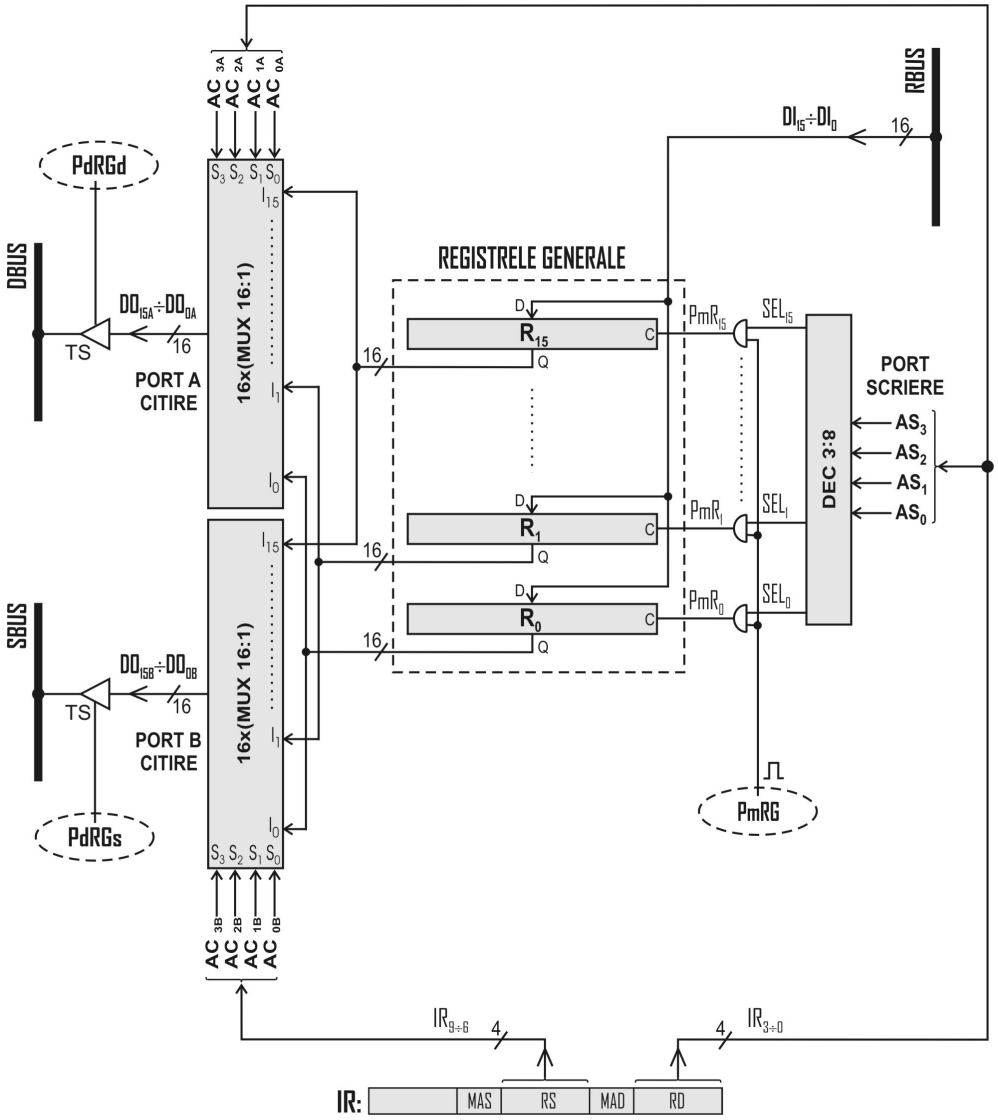


Fig 4.20..Structura multiport aferentă setului de registre generale

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 şi 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) şi reutilizat (scris) în microrutina EX (de Execuție), când în registrul destinație trebuie depus rezultatul. Pe toată durata emulării instrucțiunii, registrele sursă şi respectiv

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

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

Instrucţiunile cu un singur operand şi instrucţiunile de salt utilizează un singur registru şi anume registrul destinaţie. Schema de selecţie a registrelor generale este aceeaşi doar că, pe parcursul emulării unei asemenea instrucţiuni, va fi utilizat doar registrul destinaţie.

Instrucţiunile fără operanzi nu operează cu registrele generale. Prin urmare, schema de selecţie a registrelor generale din figura 4.20 va fi neutilizată pe parcursul emulării unei asemenea instrucţiuni. De fapt, se vor selecta fals două registre (unul sursă şi unul destinaţie) dar acestea nu vor fi utilizate deoarece, pe parcursul emulării unei asemenea instrucţiuni, nu vor fi activate niciuna dintre cele trei microcomenzi care acţionează asupra registrelor generale ($PdRG_S$, $PdRG_D$ şi respectiv PmRG).

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

Observaţii:

1. *Dacă analizăm schema bloc aferentă procesorului CISC (figura 4.1), constatăm că şi registrele speciale (FLAG, SP, T, PC, IVR, ADR, şi MDR) vor avea alocate tot câte 3 microcomenzi în microinstrucţiunea generală (similitudine perfectă cu blocul de REGISTRE GENERALE).*
   Dacă ne referim de exemplu la registrul SP, microcomenzile dedicate acestuia sunt:

   - $PdSP_S$ *–pentru emisia pe SBUS a conţinutului registrului SP*
   - $PdSP_D$ *–pentru emisia pe DBUS a conţinutului registrului SP*
   - PmSP *–pentru scrierea în registrul 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 DESTINAŢIE RBUS (fig 4.18). Similar cu registrele generale, toate registrele speciale au alocate 3 astfel de microcomenzi în microinstrucţiunea generală.*

2. *În câmpurile* SURSA SBUS *şi* SURSA DBUS *apar suplimentar microcomenzi prin care se generează constante pe* SBUS *şi respectiv* DBUS:

   - $Pd0_S$ *–emite constanta* 0 *pe* SBUS
   - $Pd0_D$ *–emite constanta* 0 *pe* DBUS
   - $Pd\text{-}1_S$ *–emite constanta* -1 *pe* SBUS
   - $Pd\text{-}1_D$ *–emite constanta* -1 *pe* DBUS

3. *Relativ la registrele T şi MDR apar suplimentar comenzile* $Pd\overline{T}_S$ *şi respectiv* $Pd\overline{MDR}_D$ *care emit (pe* SBUS *şi 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 şi V) a semnalelor de condiții generate de unitatea ALU (C_{out}, Z_R, S_R şi DCR). Această operație de setare a flag-urilor în acord cu rezultatul trebuie realizată în microrutinele de execuție aferente instrucțiunilor aritmetice şi 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 şi V) cu semnalele de condiții furnizate de unitatea ALU; va fi utilizată la instrucțiunile aritmetice.*
     - $PdCOND_L$ *–încarcă doar flag-urile Z şi S; va fi utilizată la instrucțiunile logice. Asta e, la instrucțiunile logice nu se poate obține niciodată Carry=1 şi Overflow=1; ca atare flag-urile C şi V nu se poziționează.*

5. *Pe intrarea registrului MDR regăsim de asemenea un bloc de 16 multiplexoare 2:1 (figura 4.1). Prin urmare, şi 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 operanzilor din memorie, în timp ce cu* IFCH *se realizează fetch-ul instrucţiunilor din memorie. La nivelul memoriei, microcomenzile* IFCH *şi* READ *efectuează aceeaşi operaţie: citirea informaţiilor memorate în locaţia de memorie selectată de adresa existentă în registrul ADR. Diferenţa între* IFCH *şi* READ *apare doar la nivelul procesorului (la nivelul destinatarului informaţiilor citite din memorie)*:

- *microcomanda* IFCH *încarcă infirmaţia extrasă din memorie în registrul IR (Fetch Instrucţiune).*

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

4.2.4 Proiectarea microsecvenţiatorului

Microsecvenţiatorul (automatul SEQ) generează comenzile interne BGC-ului microprogramat, comenzi prin care controlează *fetch*-ul şi execuţia microinstrucţiunilor.

A. Organigrama de funcţionare a microsecvenţiatorului

La 3.5.1 a fost prezentată organigrama de funcţionare aferentă automatului SEQ dedicat procesorului didactic. Remintim faptul că la procesorul didactic am definit 3 tipuri de microinstrucţiuni iar automatul SEQ a fost proiectat astfel încât să controleze procesarea microinstrucţiunilor, ţinând cont de tipul acestora (vezi blocurile (2) şi (7) din figura 3.7). Pentru procesorul nostru CISC am definit un singur tip de microinstrucţiune. Această microinstrucţiune generală, semnificativ mai performantă, cumulează toate capabilităţile celor 3 tipuri de microinstrucţiuni definite le procesorul didactic.

În aceste condiţii, microsecvenţiatorul aferent procesorului CISC va avea o misiune uşor simplificată deoarece nu va mai trebui să decodifice tipul microinstrucţiunii şi să iniţieze acţiuni dependente de tipul microinstrucţiunii. Prin urmare, blocurile (2) şi (7) din figura 3.7 nu vor mai exista în organigrama microsecvenţiatorului aferent procesorului CISC. Vom opera de asemenea unele simplificări şi în partea de organigramă care se referă la starea 3 a automatului SEQ. Aceste simplificări vizează reducerea numărului de ieşiri din automat şi vor fi posibile dacă introducem un *hardware* adiţional foarte simplu responsabil cu generarea acestor ieşiri (comenzi). În figura 4.21 redăm organigrama microsecvenţiatorului aferent procesorului CISC.

Această organigramă trebuie explicată prin raportare la:

-schema de principiu a unităţii de control microprogramate (fig 3.1).

-organigrama microsecvenţiatorului aferent procesorului didactic (fig 3.7)

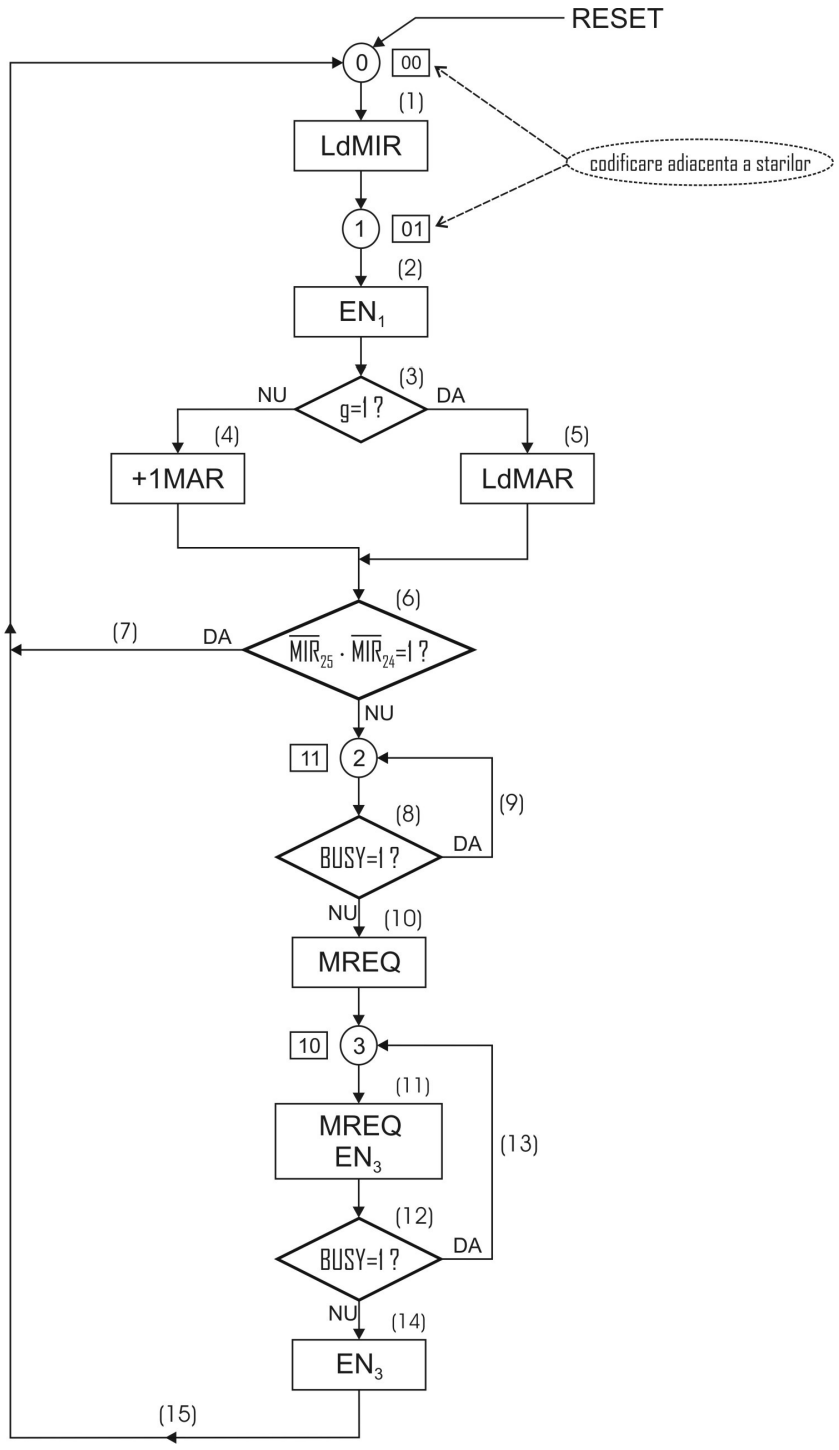


Fig 4.21..Organigrama automatului SEQ aferent procesorului CISC

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

- 2 stări (starile 0 şi 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 şi 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 şi 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 şi încărcată în registrul MIR devine astfel microinstrucțiune curentă.

În starea 1, SEQ activează comanda EN_1 (blocul **(2)** din figura 4.21) prin care validează decodificatoarele de microcomenzi (EN_1=*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 şi 4.8) şi va activa comanda +1MAR (blocul **(4)** din figura 4.21). În cazul succesorului STEP se execută deci operația:

$$MAR \leftarrow MAR+1 \qquad (4.4)$$

  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) şi dacă condiția testată este neîndeplinită. Dacă condiția testată este neîndeplinită, funcția globală de ramificație (g) va avea valoarea 0 şi 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 şi 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:

$$MAR \leftarrow MIR_{6\div0}+INDEX|_{\text{selectat de } MIR_{10\div8}} \qquad (4.5)$$

iar în cazul succesorului JUMP se execută operația:

$$MAR \leftarrow MIR_{6 \div 0} \qquad (4.6)$$

deoarece indexul selectat va avea valoarea zero (vezi $INDEX_0=0$ 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) şi dacă condiția testată este îndeplinită. Dacă condiția testată este îndeplinită, atunci funcția globală de ramificație (g) va avea valoarea 1 şi prin urmare, se va executa succesorul JUMPI (respectiv JUMP). După cum am mai menționat JUMP rezultă automat din JUMPI, dacă indexul selectat este $INDEX_0=0$.

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

- dacă $\overline{MIR_{19}} \cdot \overline{MIR_{18}} = 1$ 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 organigrama 4.21). Altfel spus, procesarea microinstrucțiunii curente s-a încheiat şi SEQ revine în starea 0 pentru *fetch*-ul următoarei microinstrucțiuni.

- dacă $\overline{MIR_{19}} \cdot \overline{MIR_{18}} = 0$ 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 şi 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 BUSY=1 (memoria este ocupată, de exemplu cu un transfer DMA anterior declanşat sau cu un ciclu de regenerare), SEQ rămâne în starea 2 (bucla notată cu **(9)** în figura 4.21) aşteptând 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)** şi 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 şi suplimentar activează comanda EN_3 (blocul **(11)** din figura 4.21). Comanda EN_3 (*enable* din starea 3)

va valida decodificatorul aferent câmpului $MIR_{19\div18}$ (câmpul OPERAŢII CU MEMORIA). Acest decodificator va activa semnalele (comenzile):

- MEMR (*MEMory Read*) dacă în $MIR_{19\div18}$ este codificată microcomanda IFCH sau microcomanda READ.

- MEMW (*MEMory Write*) dacă în $MIR_{19\div18}$ este codificată microcomanda WRITE.

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

În starea 3 SEQ testează din nou semnalul BUSY (blocul **(12)** din figura 4.21)**,** aşteptând astfel încheierea operaţiei de transfer cu memoria (bucla notată cu **(13)** în organigrama 4.21). Când BUSY "cade" în 0 (când ciclul de transfer cu memoria se încheie), SEQ mai menţine activă doar comanda EN_3 şi pe următorul impuls de tact, va reveni în starea 0 pentru *fetch*-ul următoarei microinstrucţiuni (tranziţia notată cu **(15)** în organigrama 4.21).

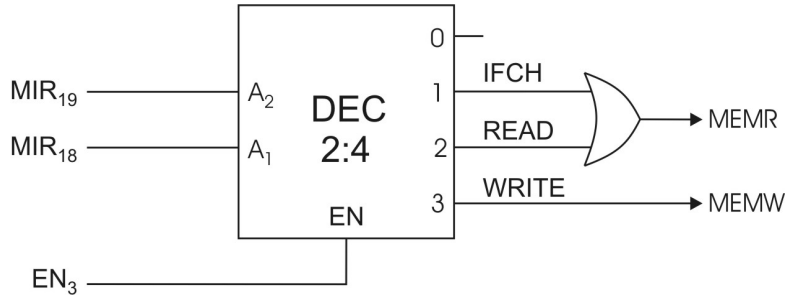


Fig 4.22 Generarea comenzilor MEMR şi MEMW pentru memorie

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

$$PmIR_3=IFCH\bullet EN_3\bullet \overline{BUSY} \qquad (4.7)$$

$$PmMDR_3=READ\bullet EN_3\bullet \overline{BUSY} \qquad (4.8)$$

Comanda $PmIR_3$ este comanda de încărcare în registrul IR a instrucţiunii extrase din memorie; se va activa dacă în microinstrucţiunea curentă este codificată microcomanda

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

Comanda $PmMDR_3$ este comanda de încărcare în registrul MDR a datelor extrase din memorie; se va activa dacă în microinstrucțiunea curentă este codificată microcomanda READ. În cadrul microprogramului de emulare, vom regăsi astfel de microinstrucțiuni 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 imediată la destinație reprezintă cod ilegal.

- pentru citirea unui operand adresat indirect sau indexat; astfel de accese vom identifica în microrutinele **FOS** (*Fetch* Operand Sursă) şi **FOD** (*Fetch* Operand Destinație) aferente modurilor de adresare indirect şi indexat (fig 4.10 şi 4.11).

- pentru citirea unui INDEX; astfel de accese vom identifica în microrutinele **FOS** (*Fetch* Operand Sursă) şi **FOD** (*Fetch* Operand Destinație) aferente modului de adresare indexat (fig 4.11) precum şi în microrutinele de execuție aferente instrucțiunilor JMP, CALL şi de salt condiționat, cu mod de adresare indexat (fig 4.6.b).).

- pentru citirea unui OFFSET; astfel de accese vom identifica în microrutinele **EX** (de Execuție) aferente instrucțiunilor JMP, CALL şi de salt condiționat, cu mod de adresare imediat (fig 4.6.a).).

Toate aceste accese vor fi inițiate prin microinstrucțiuni care au codificată microcomanda READ în câmpul OPERAȚII CU MEMORIA şi toate aceste microinstrucțiuni, la finele stării 3, vor provoca activarea comenzii $PmMDR_3$ (ecuația 4.8).

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

$$PmMDR_{global}=PmMDR+PmMDR_3 \qquad (4.9)$$

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

B. Graful de tranziții al microsecvențiatorului

Pornind de la organigrama 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 şi tranzițiile dintre stări, figurate prin arce orientate. Cele 4 stări ale automatului pot fi codificate cu ajutorul a 2 biți de stare şi codificarea pe care o propunem

este evidențiată atât în organigramă cât şi 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 ieşirile automatului (comenzi parazite). Codificarea adiacentă elimină comenzile parazite.

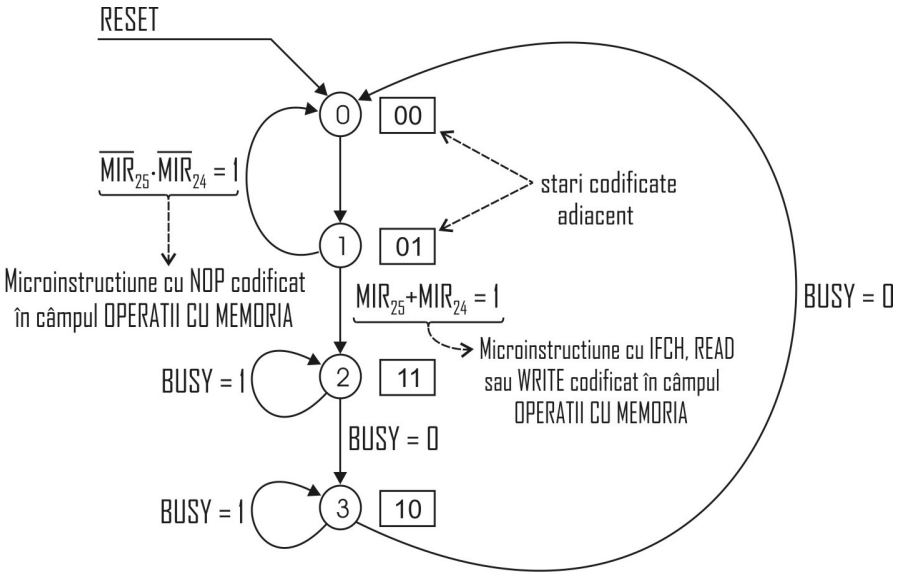


Fig 4.23 Graful de tranziții al automatului SEQ aferent procesorului CISC

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

Exemple:

1. 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 organigrama automatului (figura 4.21).

2. Tranziția (revenirea) din starea 1 în starea 0 se realizează numai dacă: $\overline{MIR_{19}} \bullet \overline{MIR_{18}} = 1$ (dacă microinstrucțiunea curentă are codificat NOP în câmpul OPERAȚII CU MEMORIA)

Să menționăm că organigrama (figura 4.21) şi graful de tranziții (figura 4.23) sunt necesare în faza de proiectare a automatului SEQ.

C. Schema bloc a microsecvenţiatorului

Schema bloc a automatului SEQ se obţine pornind de la organigrama automatului din figura 4.21. Variabilele testate în organigramă vor fi intrări în automat, iar comenzile activate în organigramă (în diferitele stări) vor fi ieşiri din automat. Automatul SEQ este un automat secvenţial sincron; trecerea din starea curentă în starea următoare se face sub controlul semnalului de tact al procesorului (CLK_P). Iniţializarea automatului se face cu ajutorul semnalului $\overline{RESET}$ care realizează de fapt iniţializarea procesorului şi respectiv calculatorului (în ansamblu). Schema bloc a automatului SEQ este redată în figura 4.24.

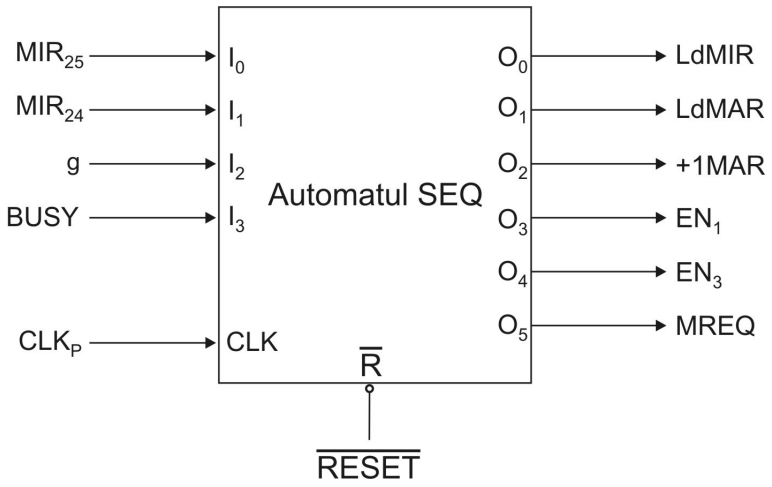


Fig 4.24 Schema bloc a automatului SEQ

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

4.2.5 Schema bloc a unităţii de control microprogramate

Un program este format dintr-o succesiune de instrucţiuni. Unitatea de control a procesorului, denumită şi bloc de generare a comenzilor (BGC), generează toate semnalele de comandă (microcomenzile) prin care se controlează procesarea instrucţiunilor (*fetch*-ul şi execuţia acestora). Structura de principiu a unitatăţii de control microprogramate a fost prezentată la 3.2 (fig 3.1), particularizată pentru procesorul didactic la 3.4.1 (fig 3.4) şi detaliată la 3.6 (fig 3.31). Particularizarea/detalierea a ţinut cont de succesorii şi de indecşii definiţi în formatul microinstrucţiunii de ramificaţie aferentă procesorului didactic.

Schema de principiu din figura 3.1 poate fi particularizată/detaliată şi pentru procesorul CISC. Această particularizare trebuie să ţină cont de formatul microinstrucţiunii generale (fig 4.18), de condiţiile testate (tabelul 4.6) şi de indecşii definiţi în cadrul microinstrucţiunii generale (tabelul 4.7). Particularizarea pentru procesorul CISC este redată în figura 4.25.

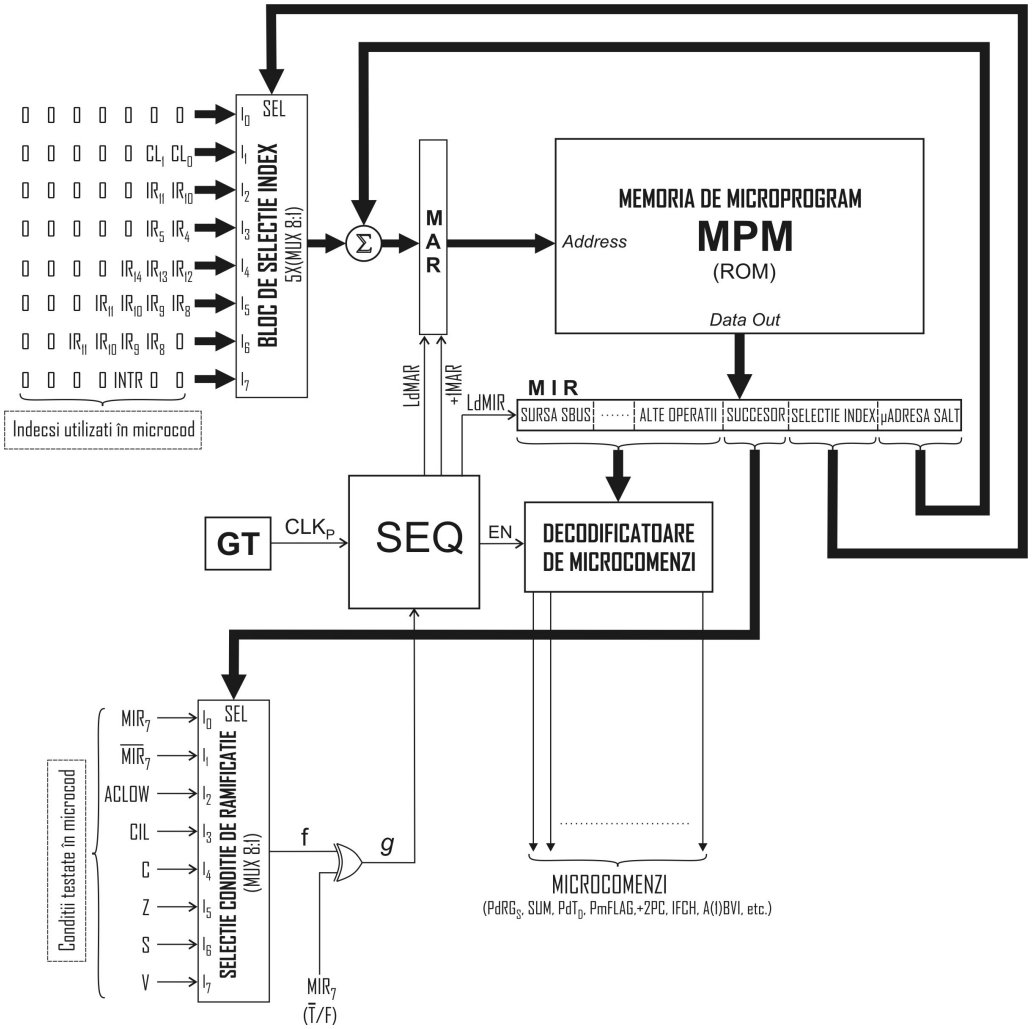


FIG 4.25 Schema bloc a unităţii de control microprogramate aferentă procesorului CISC

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

MIR –***M**icro**I**nstruction **R**egister* (registrul microinstrucțiunii). În MIR se încarcă microinstrucțiunea selectată în MPM de microadresa prezentă în MAR (*fetch*-ul microinstrucțiunii). Pentru încărcarea microinstrucțiunii în MIR, în starea 0, automatul SEQ activează comanda LdMIR (figura 4.21).

MAR –***M**icro**A**ddress **R**egister* (registrul de microadrese). MAR joacă rol de *microprogram counter* (adresează în MPM microinstrucțiunea ce urmează a fi citită şi încărcată în MIR (*fetch*-ul microinstrucțiunii). Pentru încărcarea în MAR a microadresei următoarei microinstrucțiuni, SEQ utilizează două comenzi: LdMAR (pentru succesorii JUMP şi JUMPI) şi respectiv +1MAR (pentru succesorul STEP).

MPM –***M**icro**P**rogram **M**emory* (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) şi microrutina de întrerupere (vezi organizarea microprogramului sintetizată în figura 4.17).

SEQ –***SEQ**uencer* (microsecvențiator). Automatul SEQ generează comenzile interne prin care controlează *fetch*-ul şi execuția microinstrucțiunilor; a fost prezentat la 4.2.4.

SELECȚIE CONDIȚIE DE RAMIFICAȚIE –bloc *hardware* care genereză funcția globală de ramificație g. Acest bloc are rol în execuția succesorilor (vezi explicațiile relative la blocurile **(3)**, **(4)** şi **(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) şi respectiv (4.8). Cele două tabele detaliază câmpul SUCCESOR codificat în microinstrucțiunea 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 microinstrucțiune procesată, în starea 1. Dacă g=1, atunci SEQ va genera comanda LdMAR (se execută succesorul JUMPI (respectiv JUMP dacă indexul selectat va fi $INDEX_0$=0). Asta se întâmplă când condiția testată în microinstrucțiune (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 microinstrucțiune (pe fals sau pe adevărat, după cum MIR_7 are valoarea 1 sau 0) este neîndeplinită.

BLOC DE SELECŢIE INDEX –este format din 5 multiplexoare 8:1 şi generează cei 8 indecşi definiţi la 4.2.2 (tabelul 4.7). Are deci rol în execuţia succesorilor JUMPI şi respectiv JUMP. Biţii $MIR_{10\div8}$ (câmpul SELECŢIE INDEX din microinstrucţiunea curentă) se aplică pe intrările de selecţie aferente celor 5 multiplexoare, selectând astfel indexul specificat în cadrul microinstrucţiunii. Dacă indexul selectat este diferit de zero ($INDEX_1 \div INDEX_7$) atunci se va executa într-adevăr un succesor JUMPI (vezi ecuaţia (4.5) de la 4.2.4.A). Dacă indexul selectat este $INDEX_0$=0 atunci succesorul devine JUMP (vezi ecuaţia (4.6) de la 4.2.4.A).

Observaţie: *Indecşii sunt codificaţi pe 7 biţi (vezi tabelul 4.7) deoarece câmpul* MICROADRESA DE SALT *este pe 7 biţi (fig 4.18). Cei mai semnificativi 2 biţi din index (biţii* IND_6 *şi* IND_5 *din tabelul 4.7) au valoarea zero, în toţi cei 8 indecşii definiţi. Aceşti 2 biţi nu vor necesita multiplexoare pentru selectia lor deoarece pot fi generaţi foarte simplu ca şi constante 0 (prin conectare la "0" a linilor respective). Prin urmare, doar cei mai puţin semnificativi 5 biţi din index (biţii* $IND_4 \div IND_0$ *din tabelul 4.7) revendică multiplexoare 8:1 pentru selecţie. Orice ideie care conduce la simplificări în hardware este binevenită !*

∑ –sumator pe 7 biţi care generează microadresele de salt (pentru succesorii JUMPI şi respectiv JUMP). În cazul succesorilor JUMPI se face întradevăr o operaţie de adunare (operaţia (4.5) de la 4.2.4.A). În cazul succesorilor JUMP se face o adunare cu $INDEX_0$=0 (operaţia (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 echivalenţi (tabelul 4.7).

4.2.6 Proiectarea blocurilor din componenţa unităţii de control microprogramate

În paragraful anterior au fost introdusă schema bloc a unităţii de control microprogramate (figura 4.25) şi au fost descris succint rolul fiecărui bloc din componenţa acestei unităţi. În cadrul prezentului paragraf vom trece de la descrierea succintă la proiectarea detaliată a fiecărui bloc din componenţa unităţi 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 unităţii microprogramate aferente procesorului CISC.

A. Blocul de decodificare a microcomenzilor

Microcomenzile se generează prin decodificarea câmpurilor SURSA SBUS, SURSA DBUS, OPERAŢIE ALU, DESTINAŢIE RBUS, OPERAŢII CU MEMORIA şi ALTE OPERAŢII din componenţa microinstrucţiunii generale (fig 4.18).

Câmpul OPERAŢII CU MEMORIA este un câmp de 2 biţi şi comenzile codificate în acest câmp sunt comenzi pentru memorie. Transferurile procesor-memorie sunt controlate de către automatul SEQ, în starea 3 (vezi organigrama SEQ din figura 4.21). Prin urmare, decodificatorul care generează comenzile pentru memorie trebuie validat cu semnalul de validare EN_3. Acest decodificator a fost implementat şi explicat la 4.2.4.A (fig 4.22).

Câmpurile SURSA SBUS, SURSA DBUS, OPERAȚIE ALU şi 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 şi la nevoie, poate fi procesată în ALU. În aceste transferuri pot fi specificate:

- două registre sursă (unul emis pe SBUS şi celălalt emis pe DBUS),
- o operație ALU (care se va efectua între operanzii preluați de pe SBUS şi respectiv DBUS)
- un registru destinație(în care se va încărca rezultatul obținut pe ieşirile ALU).

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

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

Toate aceste operații de transfer se desfăşoară pe durata stării 1 a automatului SEQ. Prin urmare, decodificatoarele aferente acestor câmpuri trebuie validate cu semnalul de validare EN_1, semnal activat de către automatul SEQ pe durata stării 1 (vezi organigrama SEQ din figura 4.21). Decodificatoarele aferente câmpurilor SURSA SBUS, SURSA DBUS, OPERAȚIE ALU şi DESTINAȚIE RBUS sunt prezentate în figura 4.26

Decodificatorul notat cu 1 decodifică câmpul SURSA SBUS ($MIR_{35\div32}$). Pe ieşirile decodificatorului se obțin microcomenzile (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 însemnă „nici-o sursă emisă pe SBUS". Următoarele 10 microcomenzi ($PdFLAG_S$, $PdRG_S$, $PdSP_S$, PdT_S, $Pd\overline{T}_S$, $PdPC_S$, $PdIVR_S$, $PdADR_S$, $PdMDR_S$, $PdIR[7\div0]_S$) acționează asupra registrelor conectate la SBUS (fig 4.1). Fiecare dintre cele 10 microcomenzi va emite pe SBUS datele existente în registrul specificat. Indicele „S" care apare în denumirile acestor microcomenzi specifică faptul că registrul vizat de microcomandă va fi emis pe SBUS. Este necesară această precizare deoarece toate registrele din structura procesorului CISC (fig 4.1) sunt conectate atât la SBUS cât şi la DBUS.

În figura 4.27 este evidențiat modul în care sunt conectate registrele T şi PC la SBUS. Microcomenzile evidențiate în figura 4.27 produc următoarele acțiuni:

PdT_S, -emite conținutul registrului T pe SBUS

$Pd\overline{T}_S$, -emite conținutul registrului T, inversat bit cu bit, pe SBUS

$PdPC_S$, -emite conținutul registrului PC pe SBUS

Celelalte registre din structura procesorului sunt conectate la SBUS prin scheme similare. Asta 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).

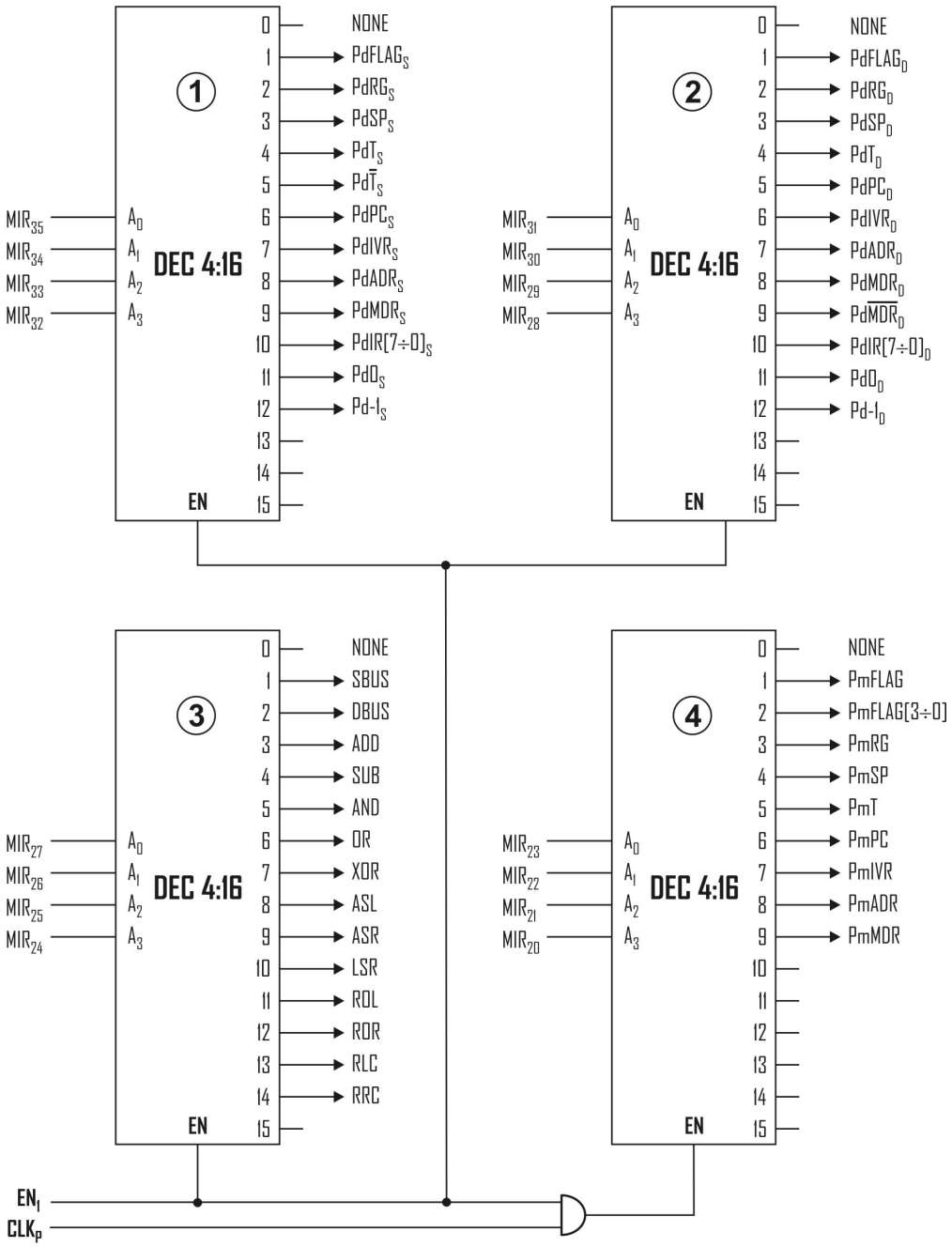


Fig 4.26 Decodificatoarele de microcomenzi aferente câmpurilor SURSA SBUS, SURSA DBUS, OPERAȚIE ALU şi DESTINAȚIE RBUS

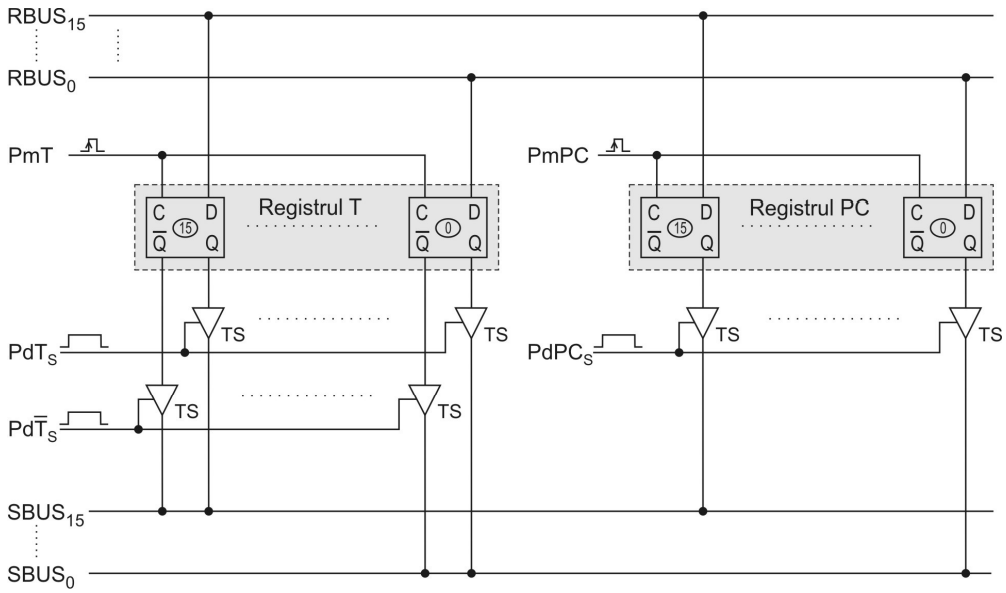


Fig 4.27 Conectarea registrelor T şi PC la SBUS

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

Decodificatorul notat cu 2 în figura 4.26 decodifică câmpul SURSA DBUS ($MIR_{31\div28}$). Pe ieşirile decodificatorului se obţin microcomenzile (de tip nivel) care emit diversele surse pe DBUS, specificate în ordinea în care au fost codificate în formatul microinstrucţiunii 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 (funcţie SAU-CABLAT pe fiecare dintre cele 16 linii de bus).

Decodificatorul notat cu 3 în figura 4.26 decodifică câmpul OPERAŢIE ALU ($MIR_{27\div24}$). Pe ieşirile decodificatorului se obţin microcomenzile (de tip nivel) dedicate unităţii ALU, specificate în ordinea în care au fost codificate în formatul microinstrucţiunii generale (fig 4.18). Să observăm că microcomenzile de deplasare şi rotire le-am codificat în câmpul OPERAŢIE ALU. Prin urmare, operaţiile de deplasare şi rotire vor fi implementate în ALU (vezi cele 2 observaţii de la 2.7.2.D).

Decodificatorul notat cu 4 în figura 4.26 decodifică câmpul DESTINAŢIE RBUS ($MIR_{23\div20}$). Pe ieşirile decodificatorului se obţin microcomenzile „*primeşte registru*", specificate în ordinea în care au fost codificate în formatul microinstrucţiunii generale (fig 4.18). Aceste microcomenzi trebuie să fie de tip impuls şi din acest motiv decodificatorul este validat cu semnalul $EN=EN_1 \bullet CLK_P$ (vezi figurile 2.18 şi 3.34 cu explicaţiile aferente).

Microcomenzile „*primeşte registru*" încarcă datele prezente pe RBUS, în registre. În cazul procesorului CISC (fig 4.1), pe RBUS vom avea întotdeauna datele obţinute pe ieşirea ALU, deoarece ALU este singura sursă conectată la SBUS.

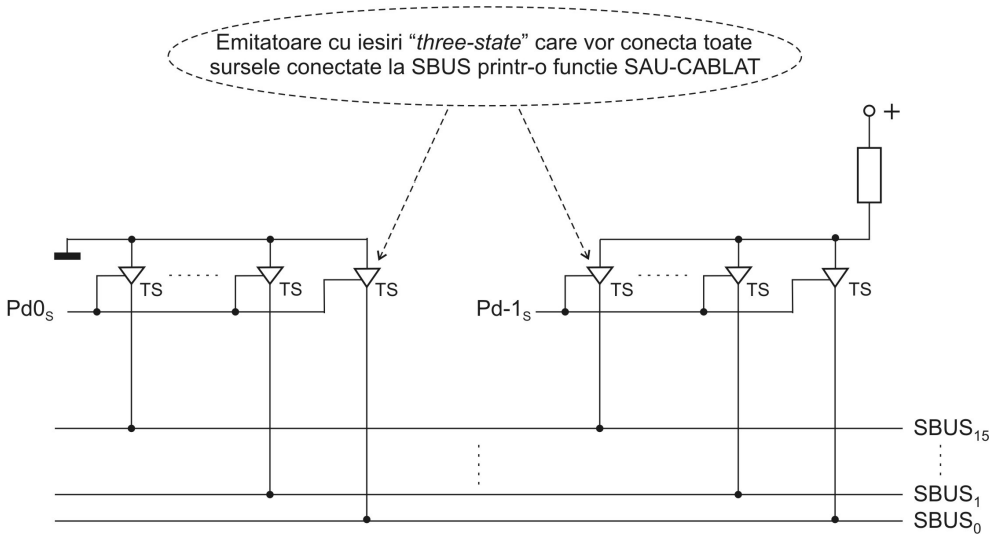


Fig 4.28 Generarea constantelor 0 şi -1 pe SBUS

În figura 4.27 este ilustrată acțiunea comenzilor PmT şi PmPC iar în figura 4.20 este ilustrată acțiunea comenzii PmRG; o astfel de microcomandă „*primeşte registru*" vom regăsi la nivelul fiecărui registru din structura procesorului.

Comenzile codificate în câmpul ALTE OPERAȚII ($MIR_{17\div14}$) 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 EN_1, iar microcomenzile sunt asignate ieşirilor 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 *Instrucțion 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 şi V în acord cu rezultatul obținut pe ieşirea 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 şi S în acord cu rezultatul obținut pe ieşirea 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*) şi decrementarea cu 2 a *stack pointer*-ului (va fi utilizată în microrutina de întrerupere)

A(0)BE, A(0)BI –resetarea bistabililor de excepție şi 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 aceleeaşi întreruperi/excepții.

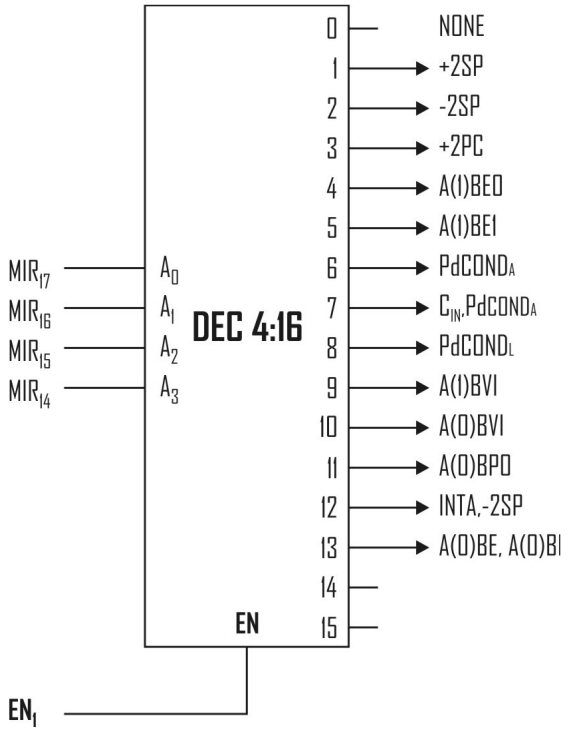


Fig 4.29 Decodificatorul de microcomenzi aferent câmpului ALTE OPERAȚII

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).

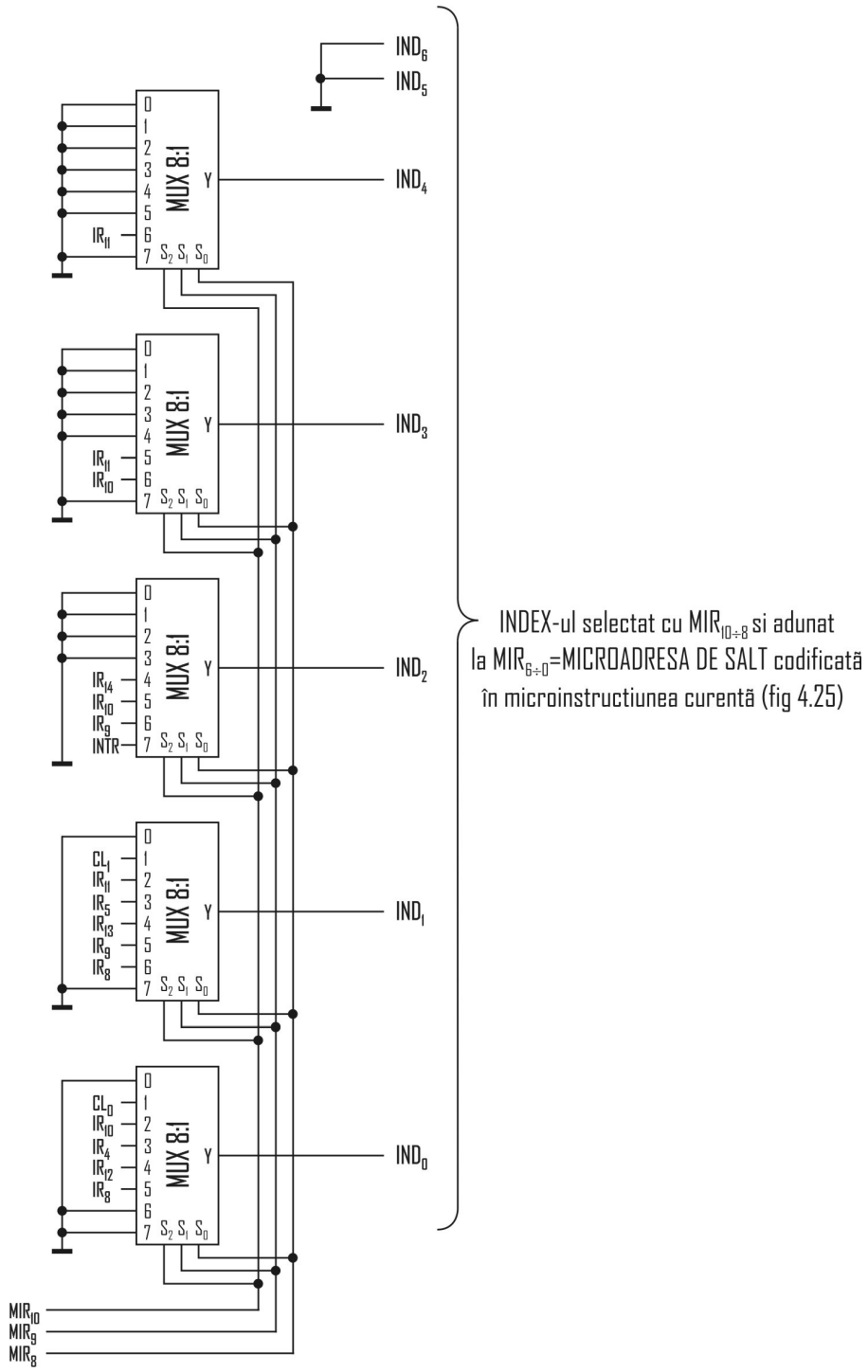


Fig 4.30 Blocul de selecție index aferent procesorului CISC

Din schema principială de organizare a microprogramului de emulare (fig 4.17) au rezultat indecşii care vor fi utilizaţi pentru implementarea ramificaţiilor indexate din cadrul microprogramului. Aceşti indecşi au fost sintetizati în tabelul 4.7. Tabelul 4.7 conţine 8 indecşi; primul index are valoarea zero iar următorii 7 sunt diferiţi de zero. Reamintim faptul că primul index ($INDEX_0$=0) permite transformarea succesorilor JUMPI (tabelul 4.6) în succesori JUMP echivalenţi (tabelul 4.8).

Strategia de proiectare aplicată în cazul blocului de selecţie index aferent procesorului didactic (paragraful 3.7.2) va fi aplicată şi aici. Diferenţele care apar sunt:

- La procesorul didactic au fost definiţi doar 3 indexi (tabelul 3.3) ceea ce a condus la un câmp SELECŢIE INDEX de 2 biţi în formatul microinstrucţiunii (fig 3.2) şi respectiv la o implementare bazată pe multiplexoare 4:1 (vezi fig 3.37).

- La procesorul CISC au fost definiţi 8 indexi (tabelul 4.7) ceea ce a condus la un câmp SELECŢIE INDEX de 3 biţi în formatul microinstrucţiunii (fig 4.18) şi respectiv la o implementare bazată pe multiplexoare 8:1 (fig 4.30).

Cei 3 biţi din componenţa câmpului SETECŢIE INDEX aferent microinstrucţiunii curente ($MIR_{10\div8}$) sunt aplicaţi pe intrările de selecţie ($S_{2\div0}$) 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 selecţie se aplică codul $MIR_{10\div8}$=001 atunci, la nivelul fiecărui multiplexor, va fi selectată intrarea **1** şi valoarea existentă pe această intrare va fi transferată la ieşirea **Y** a multiplexorului. Prin urmare, dacă $MIR_{10\div8}$=001, atunci pe ieşirile celor 5 multiplexoare se obţine $IND_{4\div0}$=0, 0, 0, 0, CL_1, CL_0. Având în vedere că biţii IND_6 şi IND_5 sunt întotdeauna zero, rezultă $IND_{6\div0}$=0, 0, 0, 0, 0, 0, CL_1, CL_0. În concluzie, dacă $MIR_{10\div8}$=001, pe ieşirile blocului de selecţie index din figura 4.30 se obţine $IND_{6\div0}$=$INDDEX_1$=0, 0, 0, 0, 0, 0, CL_1, CL_0.

În mod analog, cu ajutorul câmpului $MIR_{10\div8}$ care poate lua 8 valori distincte (de la 000_2 şi până la 111_2), sunt selectaţi şi ceilalţi indecşi definiţi în tabelul 4.7.

C. Blocul de generare a funcţiei globale de ramificaţie

Funcţia globală de ramificaţie (**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 $MIR_{10\div8}$=000, atunci INDEX-ul selectat va avea valoarea zero (tabelul 4.7) şi 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 secvenţial la următoarea microinstrucţiune.

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

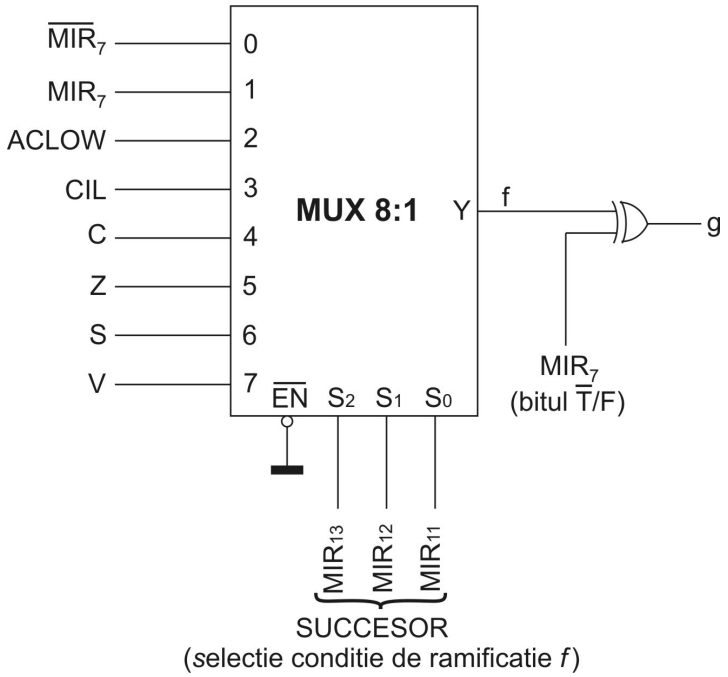


Fig 4.31 Generarea funcției globale de ramificație la procesorul CISC

La procesorul didactic a fost necesar un multiplexor 16:1 (fig 3.38) pentru a selecta condițiile cuprinse în tabelele 3.1 şi respectiv 3.2, pe ieşirea multiplexorului obținându-se funcția testată (**f**). Pe baza funcției **f** şi 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 şi 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 şi 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 şi respectiv 4.8, pe ieşirea multiplexorului obținându-se funcția testată (**f**). Pe baza funcției **f** şi a bitului MIR_7 ($\overline{T}/F$) se generează funcția globală de ramificație **g**=f$\oplus MIR_7$. Similar (cu situatia prezentată la procesorul didactic), tabelele 4.6 şi 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 şi 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 şi 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 microrutinele au fost reprezentate (simbolic) prin segmente de dreaptă iar pentru legăturile dintre microrutine au fost propuse punte de ramificație implementate prin salturi indexate. Un număr de 7 indecşi sunt evidențiați în figura 4.17, pentru implementarea legăturilor dintre microrutine. La aceşti 7 indecşi, diferiți de zero, am adăugat indexul $INDEX_0$ care are valoarea zero şi care are rolul de a transforma succesorii JUMPI (tabelul 4.6) în succesori JUMP (tabelul 4.8) Toți aceşti indecşi au fost preluați şi implementați şi în formatul microinstrucțiunii generale (vezi fig 4.18 şi tabelul 4.7). Să remarcăm faptul că inclusiv pentru implementarea punctului interuptibil se propune un salt indexat, cu $INDEX_7$=0, 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 şi 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 microproprogramului:

- au fost proiectate toate microrutinele (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 şi 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 microrutinele de execuție aferente instrucțiunilor de salt condiționat (vezi etichetele BEQ, BNE, BMI, BPL, BCS, BCC, BVS şi 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 şi explicațiile de la 3.8.2 A, 3.8.2.B şi 3.8.2.C). Singura diferență care apare, rezultă firesc din capabilitățile sporite ale microinstrucțiunii generale, care permite codificarea unui set amplu de operații concurente (vezi 4.2.2).

Tabelul 4.9 Microprogramul de emulare:

Eticheta	Adresa MPM	Micrinstructiunea	Comentarii
IFC:	00_H	PC_S SBUS NONE, PmADR, IFCH, +2PC, IF ACLOW JUMP PWFAIL else STEP	;ADR←PC (adresa de *fetch*), IR←MEM|$_{ADR}$ (fetch instructiune), ;PC←PC+2, test ACLOW (punct de ramifictie în doua directii)
	01_H	IF NCIL JUMPI A+(CL_1, CL_0) else STEP	;test CIL (cod ilegal), punct de ramificatie în 2 directii
ILLEGAL:	02_H	A(1)BE_1, JUMP INT	;BE_1←1 (pentru generarea vectorului de exceptie aferent), salt la ;**microrutina de întrerupere** (salt la eticheta INT)
PWFAIL:	03_H	A(1)BE_0, JUMP INT	;BE_0←1 (pentru generarea vectorului de exceptie aferent), salt la ;**microrutina de întrerupere** (salt la eticheta INT)
INT:	04_H	SP_S SBUS NONE, PmADR,(INTA, -2SP), STEP	;ADR←SP←SP-2, INTA=1 (achitare întrerupere; prin achitare se va ;reseta si bistabilul de exceptie sau de întrerupere setat)
	05_H	$FLAG_S$ SBUS NONE, PmMDR, WRITE, -2SP, STEP	;MEM|$_{ADR}$←FLAG (stiva←FLAG), SP←SP-2
	06_H	SP_S SBUS NONE, PmADR, STEP	;ADR←SP
	07_H	PC_S SBUS NONE, PmMDR, WRITE, STEP	;MEM|$_{ADR}$←PC (stiva←PC)
	08_H	IVR_S SBUS NONE, PmADR, READ, STEP	;ADR←IVR (ADR←vector de întrerupere/exceptie), ;MDR←MEM|$_{ADR}$ (MDR←adresa start *handler*)
	09_H	MDR_S SBUS NONE, PmPC, (A(0)BE, A(0)BI), JUMP IFC	;PC←MDR (PC←adresa start *handler*), BE←0, BI←0, ;salt la **microrutina *Instruction Fetch*** (salt la eticheta IFC)
A:	$0A_H$	JUMPI FOSAD+(IR_{11}, IR_{10})	;punct de ramificatie pe microrutinele ***Fetch* operand sursa** (clasa A)
B:	$0B_H$	JUMPI FODADB+(IR_5, IR_4)	;punct de ramificatie pe **setul B de microrutine *Fetch* operand** **;destinatie** (microrutinele FOD aferente clasei B)
C:	$0C_H$	JUMPI BEQ+(IR_{11}÷IR_8, 0)	;punct de ramificatie pe **microrutinele de executie** aferente clasei C
D:	$0D_H$	JUMPI CLC+(IR_{11}÷IR_8, 0)	;punct de ramificatie pe **microrutinele de executie** aferente clasei D
FOSAD:	$0E_H$	RGs SBUS NONE, PmT, JUMPI FODADA+(IR_5, IR_4)	;T←operand sursa, punct de ramificatie pe **setul A de microrutine** **;*Fetch* operand destinatie** (microrutinele FOD aferente clasei A)

FOSAM:	$0F_H$	PC_S SBUS NONE, PmADR, READ, +2PC, JUMP FOSEND	;ADR←PC, MDR←MEM|$_{ADR}$ (MDR←operand sursa), PC←PC+2
FOSAI:	10_H	RG_S SBUS NONE, PmADR, READ, JUMP FOSEND	;ADR←RG_S (ADR←adresa operand sursa), ;MDR←MEM|$_{ADR}$ (MDR←operand sursa)
FOSAX:	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)
FOSEND:	13_H	MDR_S SBUS NONE, PmT, JUMPI FODADA+(IR_5, IR_4)	;T←operand sursa, punct de ramificatie pe **setul A de microrutine** **;*Fetch* operand destinatie** (microrutinele FOD aferente clasei A)
FODADA:	14_H	NONE DBUS RG_D, PmMDR, JUMPI MOV+(IR_{14}÷IR_{12})	;MDR←operand destinatie, punct de ramificatie pe **microrutinele de** **;executie** aferente clasei A
FODAMA:	15_H	JUMP ILLEGAL	; adresare imediata (AM) la destinatie este CIL (cod ilegal)
FODAIA:	16_H	NONE DBUS RG_D, PmADR, READ, JUMPI MOV+(IR_{14}÷IR_{12})	;ADR←adresa operand destinatie, MDR←operand destinatie, punct ;de ramificatie pe **microrutinele de executie** aferente clasei A
FODAXA:	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, JUMPI MOV+(IR_{14}÷IR_{12})	;ADR←MDR+RG_D (ADR←index destinatie+registru destinatie), ;MDR←MEM|$_{ADR}$ (MDR←operand destinatie), punct de ramificatie ;pe **microrutinele de executie** aferente clasei A
FODADB:	19_H	NONE DBUS RG_D, PmMDR, JUMPI CLR+(IR_{11}÷IR_8)	;MDR←operand destinatie, punct de ramificatie pe **microrutinele de** **;executie** aferente clasei B
FODAMB:	$1A_H$	JUMP ILLEGAL	;adresare imediata (AM) la destinatie este CIL (cod ilegal)
FODAIB:	$1B_H$	NONE DBUS RG_D, PmADR, READ, JUMPI MOV+(IR_{14}÷IR_{12})	;ADR←adresa operand destinatie, MDR←operand destinatie, punct ;de ramificatie pe **microrutinele de executie** aferente clasei B
FODAXB:	$1C_H$	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, JUMPI CLR+(IR_{11}÷IR_8)	;ADR←MDR+RG_D (ADR←index destinatie+registru destinatie), ;MDR←MEM|$_{ADR}$ (MDR←operand destinatie), punct de ramificatie ;pe **microrutinele de executie** aferente clasei B

MOV:	$1E_H$	T_S SBUS NONE, PmMDR, JUMPI WRD+(IR_5, IR_4)	;MDR⟵T (MDR⟵operand sursa), salt indexat la WRD (salt indexat ;la microrutina de scriere rezultat la destinatie)
ADD:	$1F_H$	T_S SUM MDR_D, PmMDR, $PdCOND_A$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵ MDR+T, pozitionare *flag*-uri C/Z/S/V, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
SUB:	20_H	$\overline{T}_S$ SUM MDR_D, PmMDR, (C_{IN}, $PdCOND_A$), JUMPI WRD+(IR_5, IR_4)	;MDR⟵MDR-T, pozitionare *flag*-uri C/Z/S/V, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
CMP:	21_H	$\overline{T}_S$ SUM MDR_D, (C_{IN}, $PdCOND_A$), JUMPI IFC+(INTR, 0, 0)	;MDR-T doar pentru pozitionare *flag*-uri C/Z/S/V, punct interuptibil
AND:	22_H	T_S AND MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵MDR AND T, pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
OR:	23_H	T_S OR MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵MDR OR T, pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
XOR:	24_H	T_S XOR MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵MDR XOR T, pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
CLR:	25_H	NONE DBUS 0_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵0, pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
NEG:	26_H	NONE DBUS $\overline{MDR}_D$, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵ $\overline{MDR}$, pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
INC:	27_H	0_S SUM MDR_D, PmMDR, (C_{IN}, $PdCOND_A$), JUMPI WRD+(IR_5, IR_4)	;MDR⟵MDR+1, pozitionare *flag*-uri C/Z/S/V, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
DEC:	28_H	-1_S SUM MDR_D, PmMDR, $PdCOND_A$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵MDR-1, pozitionare *flag*-uri C/Z/S/V, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
ASL:	29_H	NONE ASL MDR_D, PmMDR, $PdCOND_A$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵ASL(MDR), pozitionare *flag*-uri C/Z/S/V, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
ASR:	$2A_H$	NONE ASR MDR_D, PmMDR, $PdCOND_A$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵ASR(MDR), pozitionare *flag*-uri C/Z/S/V, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
LSR:	$2B_H$	NONE LSR MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵LSR(MDR), pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
ROL:	$2C_H$	NONE ROL MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR⟵ROL(MDR), pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)

ROR:	$2D_H$	NONE ROR MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR←ROR(MDR), pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
RLC:	$2E_H$	NONE RLC MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR←RLC(MDR), pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
RRC:	$2F_H$	NONE RRC MDR_D, PmMDR, $PdCOND_L$, JUMPI WRD+(IR_5, IR_4)	;MDR←RRC(MDR), pozitionare *flag*-uri Z/S, salt indexat la WRD ;(salt indexat la microrutina de scriere rezultat la destinatie)
PUSH:	30_H	SP_S SBUS NONE, PmADR, WRITE, -2SP, JUMPI IFC+(INTR, 0, 0)	;ADR←SP-2 (-2SP de tip nivel pt. a încarca în ADR valoarea SP-2), ;MEM|$_{ADR}$←MDR (scriere în stiva)
POP:	31_H	ADR_S SBUS NONE, PmT, STEP	;salvare ADR în T pt. a face POP functional cu toate modurile de ;adresare
	32_H	SP_S SBUS NONE, PmADR, READ, STEP	;ADR←SP, MDR←MEM|$_{ADR}$ (citire locatie din vârful stivei)
	33_H	T_S, SBUS NONE, PmADR, +2SP, JUMPI WRD+(IR_5, IR_4)	;ADR←T (restaurare ADR din T), SP←SP+2 (noul vârf al stivei), ;salt indexat la WRD (POP functional cu toate modurile de adresare !)
WRD:	34_H	MDR_S SBUS NONE, PmRG, JUMPI IFC+(INTR, 0, 0)	;RG←MDR (Registru destinatie←Rezultat) -adresare directa (AD)
	35_H	JUMP ILLEGAL	;destinatie cu adresare imediata (AM) este CIL (cod ilegal)
	36_H	WRITE, JUMPI IFC+(INTR, 0, 0)	;MEM|$_{ADR}$←MDR (Memorie←Rezultat) -adresare indirecta (AI) ;punct interuptibil
	37_H	WRITE, JUMPI IFC+(INTR, 0, 0)	;MEM|$_{ADR}$←MDR (Memorie←Rezultat) -adresare indexata (AX) ;punct interuptibil
BEQ:	38_H	IF Z JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca Z=1: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	39_H	JUMPI IFC+(INTR, 0, 0)	;daca Z=0: punct interuptibil (conditie de salt neîndeplinita)
BNE:	$3A_H$	IF NZ JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca Z=0: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	$3B_H$	JUMPI IFC+(INTR, 0, 0)	;daca Z=1: punct interuptibil (conditie de salt neîndeplinita)
BMI:	$3C_H$	IF S JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca S=1: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	$3D_H$	JUMPI IFC+(INTR, 0, 0)	;daca S=0: punct interuptibil (conditie de salt neîndeplinita)

BPL:	$3E_H$	IF NS JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca S=0: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	$3F_H$	JUMPI IFC+(INTR, 0, 0)	;daca S=1: punct interuptibil (conditie de salt neîndeplinita)
BCS:	40_H	IF C JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca C=1: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	41_H	JUMPI IFC+(INTR, 0, 0)	;daca C=0: punct interuptibil (conditie de salt neîndeplinita)
BCC:	42_H	IF NC JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca C=0: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	43_H	JUMPI IFC+(INTR, 0, 0)	;daca C=1: punct interuptibil (conditie de salt neîndeplinita)
BVS:	44_H	IF V JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca V=1: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	45_H	JUMPI IFC+(INTR, 0, 0)	;daca V=0: punct interuptibil (conditie de salt neîndeplinita)
BVC:	46_H	IF NV JUMPI JMPAD+(IR_5, IR_4) else STEP	;daca V=0: salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	47_H	JUMPI IFC+(INTR, 0, 0)	;daca V=1: punct interuptibil (conditie de salt neîndeplinita)
JMP:	48_H	JUMPI JMPAD+(IR_5, IR_4)	;salt la executie JMP cu AM (salt direct), JMP cu AD (CIL), ;JMP cu AI (salt indirect), JMP cu AX (salt indexat)
	49_H	NONE	
CALL:	$4A_H$	JUMPI CALLAD+(IR_5, IR_4)	;salt indexat la executie: CALL cu AD (CIL), CALL cu AM (call direct), ;CALL cu AI (call indirect), CALL cu AX (call indexat)
CALLAD:	$4B_H$	JUMP ILLEGAL	;CALL cu adresare directa este cod ilegal (CIL)
CALLAM:	$4C_H$	PC_S SBUS NONE, PmADR, READ, +2PC, JUMP CALL2	;ADR←PC, MDR←MEM $\vert_{ADR}$ (MDR←Offset), PC←PC+2
CALLAI:	$4D_H$	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←PC, MDR←MEM $\vert_{ADR}$ (MDR←Index), PC←PC+2
	$4F_H$	MDR_S SUM RG_D, PmT, JUMP CALL3	;T←MDR+RG_D (T←adresa de call; adresa de call=RG_D+index)
CALL2:	50_H	MDR_S SUM PC_D, PmT, STEP	;T←MDR+PC (T←adresa de call; adresa de call=PC+Offset)

CALL3:	51_H	SP_S SBUS NONE, PmADR, -2SP, STEP	;ADR←SP-2, (-2SP comanda de tip nivel astfel încât în ADR sa se ;încarce valoarea obtinuta în SP dupa decrementare)
	52_H	PC_S SBUS NONE, PmMDR, WRITE, STEP	;MEM|$_{ADR}$←PC (Stiva←PC; PC-ul de revenire salvat în stiva)
	53_H	T_S SBUS NONE, PmPC, JUMPI IFC+(INTR, 0, 0)	;PC←T (PC←adresa de call), punct interuptibil
JMPAD:	54_H	JUMP ILLEGAL	;JMP cu adresare directa este cod ilegal (CIL)
JMPAM:	55_H	PC_S SBUS NONE, PmADR, READ, +2PC, JUMPAM2	;ADR←PC, MDR←MEM|$_{ADR}$ (MDR←Offset), PC←PC+2
JMPAI:	56_H	NONE DBUS RG_D, PmPC, JUMPI IFC+(INTR, 0, 0)	;PC←RG_D (PC←adresa de salt), punct interuptibil
JMPAX:	57_H	PC_S SBUS NONE, PmADR, READ, +2PC, STEP	;ADR←PC, MDR←MEM|$_{ADR}$ (MDR←Index), PC←PC+2
	58_H	MDR_S SUM RG_D, PmPC, JUMPI IFC+(INTR, 0, 0)	;PC←MDR+RG_D (PC←adresa de salt; adresa de salt=RG_D+index)
JMPAM2:	59_H	MDR_S SUM PC_D, PmPC, JUMPI IFC+(INTR, 0, 0)	;PC←MDR+PC (PC←adresa de salt; adresa de salt=PC+Offset)
CLC:	$5A_H$	$FLAG_S$ AND $IR[7\div0]_D$, PmFLAG, JUMPI IFC+(INTR, 0, 0)	;FLAG←FLAG AND IR[7÷0] (IR[7÷0=Masca), punct interuptibil ;(bitii pe 0 în masca desemneaza *flag*-urile ce vor fi resetate)
	$5B_H$	NONE	
SEC:	$5C_H$	$FLAG_S$ OR $IR[7\div0]_D$, PmFLAG, JUMPI IFC+(INTR, 0, 0)	;FLAG←FLAG OR IR[7÷0] (IR[7÷0=Masca), punct interuptibil ;(bitii pe 1 în masca desemneaza *flag*-urile ce vor fi setate)
	$5D_H$	NONE	
NOP:	$5E_H$	JUMPI IFC+(INTR, 0, 0)	;punct interuptibil
	$5F_H$	NONE	
HALT:	60_H	A(0)BPO	;BPO←0 (resetare bistabil pornire-oprire pt. blocare tact procesor)
	61_H	NONE	
EI:	62_H	A(1)BVI, JUMPI IFC+(INTR, 0, 0)	; BVI←1 (setare bistabil de validare a intreruperilor), punct interupt.
	63_H	NONE	
DI:	64_H	A(0)BVI, JUMP IFC	;BVI←0 (resetare bistabil de validare a intreruperilor), salt la IFC
	65_H	NONE	
PUSH PC:	66_H	SP_S SBUS NONE, PmADR, -2SP, STEP	;ADR←SP-2 (-2SP de tip nivel pt. a încarca în ADR valoarea SP-2)

	67_H	PC_S SBUS NONE, PmMDR, WRITE, JUMPI IFC+(INTR, 0, 0)	;MEM|$_{ADR}$←PC (Stiva←PC), punct interuptibil
POP PC:	68_H	SP_S SBUS NONE, PmADR, READ, STEP	;ADR←SP, MDR←MEM|$_{ADR}$ (MDR←continut locatie din vârful stivei)
	69_H	MDR_S SBUS NONE, PmPC, +2SP, JUMPI IFC+(INTR, 0, 0)	;PC←MDR, SP←SP-2, punct interuptibil
PUSH FLAG:	$6A_H$	SP_S SBUS NONE, PmADR, -2SP, STEP	;ADR←SP-2 (-2SP de tip nivel pt. a încarca în ADR valoarea SP-2)
	$6B_H$	$FLAG_S$ SBUS NONE, PmMDR, WRITE, JUMPI IFC+(INTR, 0, 0)	;MEM|$_{ADR}$←FLAG (Stiva←FLAG), punct interuptibil
POP FLAG:	$6C_H$	SP_S SBUS NONE, PmADR, READ, STEP	;ADR←SP, MDR←MEM|$_{ADR}$ (MDR←continut locatie din vârful stivei)
	$6D_H$	MDR_S SBUS NONE, PmFLAG, +2SP, JUMPI IFC+(INTR, 0, 0)	;FLAG←MDR, SP←SP-2, punct interuptibil
RET:	$6E_H$	SP_S SBUS NONE, PmADR, READ, STEP	;ADR←SP, MDR←MEM|$_{ADR}$ (MDR←continut locatie din vârful stivei)
	$6F_H$	MDR_S SBUS NONE, PmPC, +2SP, JUMPI IFC+(INTR, 0, 0)	;PC←MDR, SP←SP-2, punct interuptibil
IRET:	70_H	SP_S SBUS NONE, PmADR, READ, STEP	;ADR←SP, MDR←MEM|$_{ADR}$ (MDR←continut locatie din vârful stivei)
	71_H	MDR_S SBUS NONE, PmPC, +2SP, STEP	;PC←MDR, SP←SP-2
	72_H	SP_S SBUS NONE, PmADR, READ, STEP	;ADR←SP, MDR←MEM|$_{ADR}$ (MDR←continut locatie din vârful stivei)
	73_H	MDR_S SBUS NONE, PmFLAG, +2SP, JUMPI IFC+(INTR, 0, 0)	;FLAG←MDR, SP←SP-2, punct interuptibil

Tabelul 4.9 Microprogramul de emulare aferent procesorului CISC

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 şi 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 şi 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 şi 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 şi 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 parcurse 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 şi 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 ieşirile decodificatorului instrucțiunii se vor activa semnalele care descriu tipul instrucțiunii. Printre aceste semnale, vom regăsi şi semnalele CL_1, CL_0 , care codifică clasa din care face parte instrucțiunea. Este evident că următoarea microinstrucțiune (micrfoinstrucțiunea de la microadresa 01_H) beneficiază de semnale CL_1, CL_0 stabile şi corecte şi 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) şi-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)

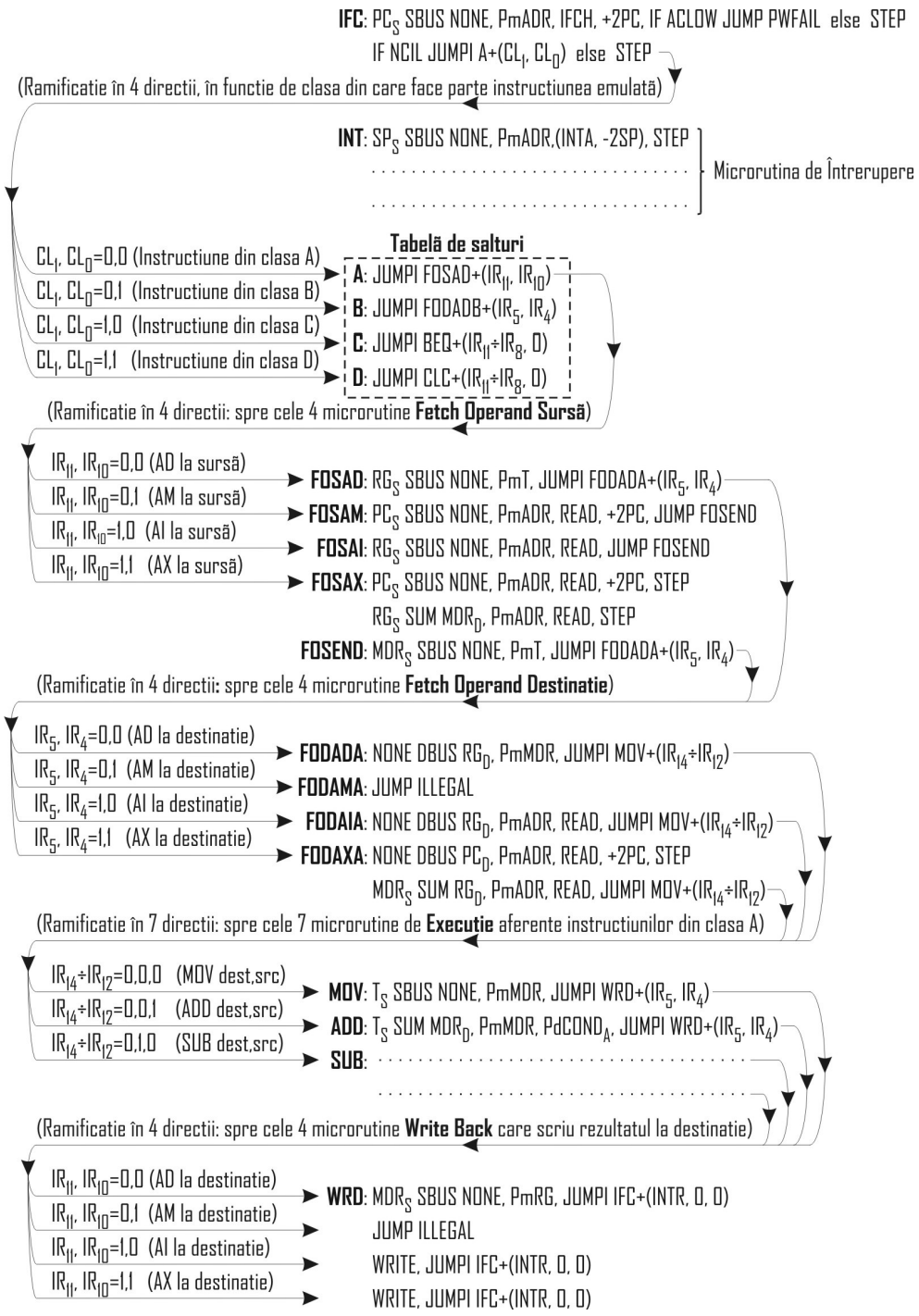


Fig 4.32 Mecanismul de selecţie a microrutinelor în cadrul procesului de emulare a unei instrucţiuni din clasa A

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 şi 4.11. Legătura între microrutina **IF** şi cele 4 microrutine **FOS** este implementată prin două salturi indexate succesive. Primul salt este JUMPI A+(CL_1, CL_0) şi 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+(CL_1, CL_0) realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa A, atunci CL_1, CL_0=0,0 şi succesorul JUMPI A+(CL_1, CL_0) va realiza saltul în prima locație a tabelei de salturi (la microadresa $0A_H$). Aici regăsim cel de-al doilea salt indexat, JUMPI FOSAD+(IR_{11}, IR_{10}), 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) şi-l va încărca în procesor (în registrul MDR). Există 3 microrutine **FOD_A** (câte una pentru fiecare mod de adresare <u>acceptat</u> 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 şi 4.11.

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

   Legătura dintre microrutinele **FOS** şi microrutinele **FOD_A** este realizată prin saltul indexat JUMPI FODADA+(IR_5, IR_4) pe care-l regăsim în microprogram la microadresele $0E_H$ şi 13_H (vezi figura 4.32 şi tabelul 4.9).

4. **Microrutina EX_A** (de Execuție) în care se efectuează operația (revendicată de instrucțiune) asupra celor 2 operanzi, se generează rezultatul şi se memorează acest rezultat la destinație (se suprascrie operandul destinație). Există 7 instrucțiuni în clasa A: (MOV, ADD, SUB, CMP, AND, OR şi 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) şi respectiv XOR (microadresa 24_H). Microinstrucțiunile de la cele 7 etichete precizate, efectuează operația aritmetico-logică (revendicată de instrucțiune), poziționează *flag*-urile de condiții în acord cu rezultatul obținut (excepție face instrucțiunea MOV care nu poziționează *flag*-urile) şi execută saltul indexat JUMPI WRD+(IR_5, IR_4). 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 şi instrucțiune ilegală. Din acest motiv, la microadresa 35_H, regăsim saltul necondiționat: JUMP ILLEGAL.

La celelalte 3 microadrese (34_H, 36_H şi 37_H) se realizează scrierea rezultatului la destinație. Din acest motiv, în figura 4.32 le-am denumit microrutine *write back*. Aceste microrutine *write back* conțin câte o singură microinstrucțiune şi reprezintă de fapt partea finală a microrutinei $\mathbf{EX_A}$. Pe cele 3 microinstrucțiuni care realizează operația *write back* este implementat şi punctul interuptibil. Asta e, s-a finalizat execuția instrucțiunii şi la sfârşitul fiecărei instrucțiuni trebuie implementat punctul interuptibil!

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 AI şi AX, rezultatul se scrie în memorie, la adresa existentă în rgistrul ADR (microcomanda WRITE de la microadresele 36_H şi respectiv 37_H). Cum în microrutina anterioară (în microrutina $\mathbf{FOD_A}$), 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 interuptibil de la sfârşitul instrucțiunii (vezi microadresele 34_H, 36_H şi 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 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 INTR=1. Prin urmare, dacă cel puțin un periferic a activat cererea de întrerupere şi întreruperile sunt validate, se trece la microrutina de întrerupere. Aceasta va suspenda temporar *task*-ul (programul) curent şi va lansa în execuție *handler*-ul de întrerupere aferent perifericului care a solicitat întreruperea. *Handler*-ul va face perifericului solicitant, serviciul solicitat. Evident că şi 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 şi respectiv destinație, poziționează *flag*-urile de condiții în acord cu rezultatul comparației (scăderii), dar <u>nu depune rezultatul la destinație</u>! În fond, prin aceasta se deosebeşte de instrucțiunea SUB (vezi microadresa 21_H din tabelul 4.19).

Legătura dintre microrutinele $\mathbf{FOD_A}$ şi microrutinele $\mathbf{EX_A}$ este implementată prin saltul indexat JUMPI MOV+(IR_{14}÷IR_{12}) (vezi microadresele 14_H, 16_H şi 18_H în tabelul 4.9 şi 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 microrutinele de emulare în extenso.

Pentru emularea unei instrucțiuni din clasa B se parcurge o succesiune de 3 microrutine. Microrutinele parcurse, precum şi legăturile dintre aceste microrutine, 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 parcurge microrutina **FOS** (*Fetch* Operand Sursă). Cele 3 microrutine parcurse 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 şi la instrucțiunile din clasa A, microrutina EX_B se încheie cu aceeaşi operație *write back*. În figurile 4.32 şi 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 microrutinele $\mathbf{FOD_A}$, vom avea evident 3 microrutine $\mathbf{FOD_B}$ (câte una pentru fiecare mod de adresare <u>viabil</u> la destinație. Acestea se regăsesc în tabelul 4.19 la microadresele: 19_H (eticheta FODADB), $1B_H$ (eticheta FODAIB) şi respectiv $1C_H$ (eticheta FODAXB). Menționăm încă o dată că adresarea imediată la destinație este

consideată ilegală (vezi 4.1.3.B). Din acest motiv nu există o microrutină **FOD$_B$** cu adresare imediată şi prin urmare, la eticheta FODAMB (microadresa 1A$_H$) regăsim o microinstrucțiune de salt (JUMP ILLEGAL) care face trimitere la tratarea excepției CIL (instrucțiune ilegală !).

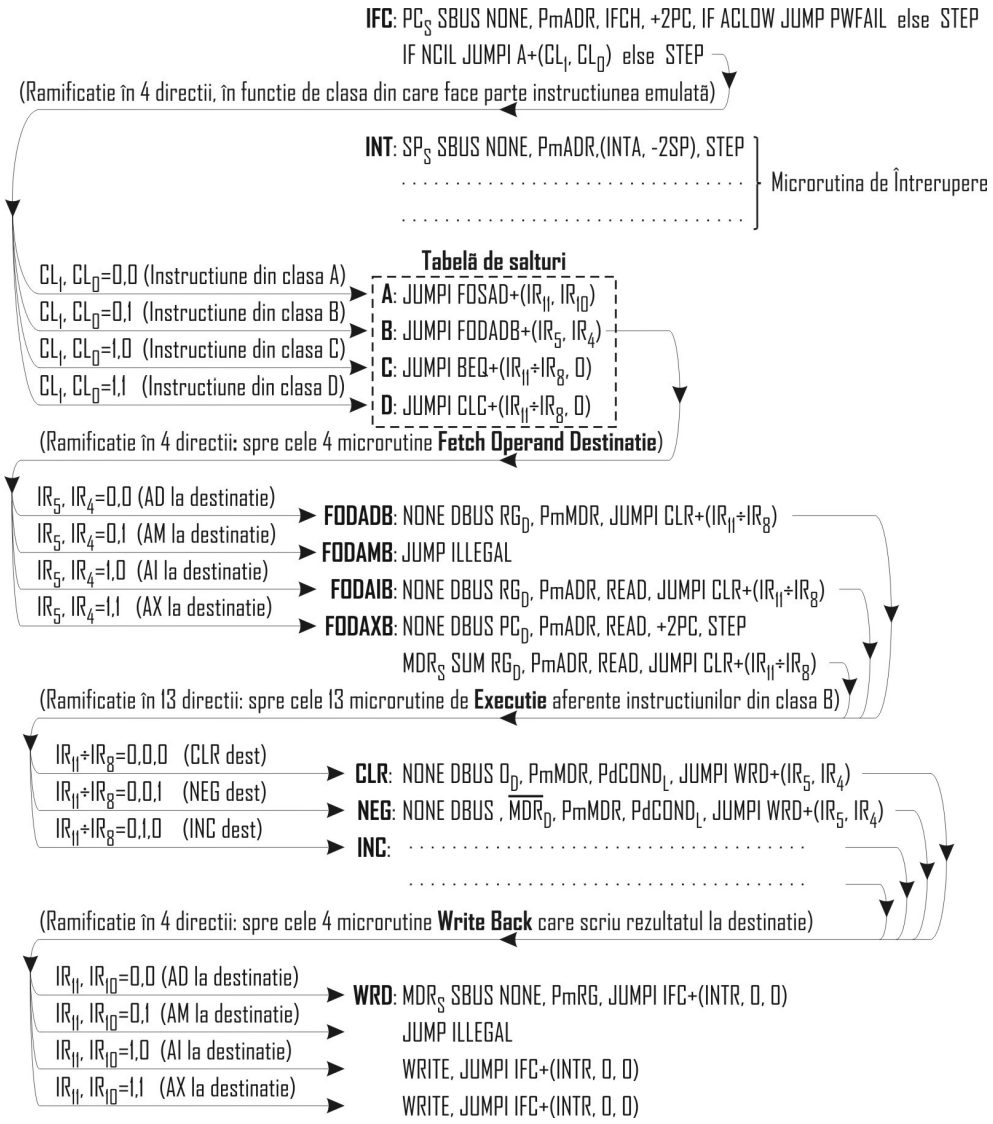


Fig 4.33 Mecanismul de selecție a microrutinelor în cadrul procesului de emulare a unei instrucțiuni din clasa B

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

JUMPI MOV+(IR_{14}÷IR_{12}), în cazul microrutinelor **FOD_A**.

JUMPI CLR+(IR_{11}÷IR_8), în cazul microrutinelor **FOD_B**.

În concluzie, dacă facem abstracție de diferența care apare la nivelul succesorilor utilizați, cele 4 microrutine **FOD_B** reprezintă copii identice ale microrutinelor **FOD_A**. Am decis implementarea a două seturi identice de microrutine **FOD** (**FOD_A** şi **FOD_B**) pentru a simplifica intrarea în microrutinele **EX_A** şi respectiv **EX_B**. Dacă am fi utilizat acelaşi set de microrutine **FOD** pentru ambele clase de instrucțiuni, atunci, la ieşirea din microrutinele **FOD,** ar fi trebuit implementate două mecanisme de ramificație succesive: primul în două direcții (clasa A şi respectiv B) şi al doilea pe microrutina aferentă instrucțiunii din cadrul clasei. O astfel de ramificație prin două salturi indexate succesive 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** şi cele 4 microrutine **FOD_B** (3 viabile şi a patra reprezentând instrucțiune ilegală) este implementată prin două salturi indexate succesive. Primul salt este JUMPI A+(CL_1, CL_0) şi 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+(CL_1, CL_0) realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa B, atunci CL_1, CL_0=0,1 şi succesorul JUMPI A+(CL_1, CL_0) 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+(IR_5, IR_4), 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 şi 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).

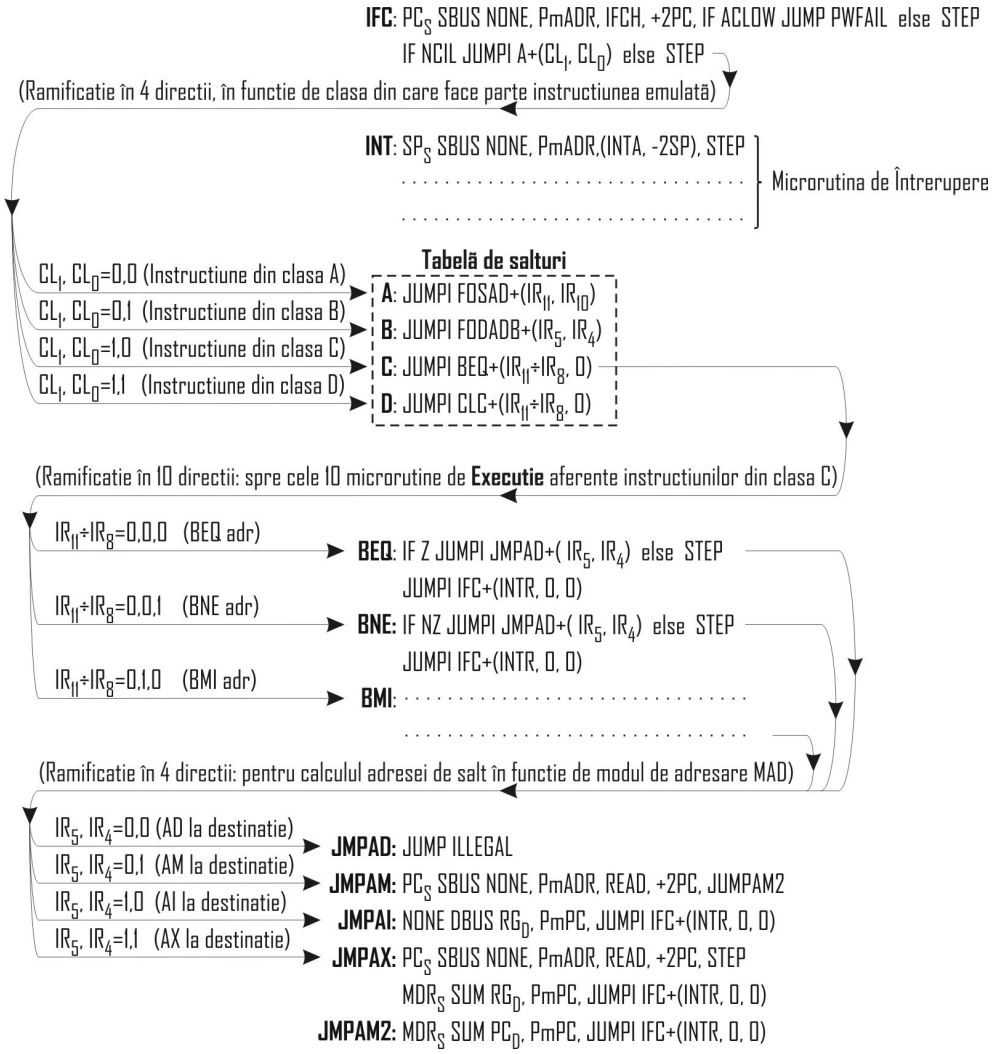


Fig 4.34 Mecanismul de selecție a microrutinelor în cadrul procesului de emulare a unei instrucțiuni de salt

Legătura între microrutina **IF** şi cele 10 microrutine **EX$_C$** este implementată prin două salturi indexate succesive. Primul salt este JUMPI A+(CL_1, CL_0) şi 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+(CL_1, CL_0) realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa C, atunci CL_1, CL_0=1,0 şi succesorul JUMPI A+(CL_1, CL_0) va realiza saltul în cea de a treia locație a tabelei de salturi (la microadresa

$0C_H$). Aici regăsim o altă microinstrucțiune de salt indexat, JUMPI BEQ+(IR_{11}÷IR_8, 0), care implementează legătura spre cele 10 microrutine **EX_C** (vezi fig 4.34).

Microrutinele **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 microrutinele **EX_C**, se calculează adreasa de salt (respectiv de *call* la instrucțiunea CALL) şi 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 neândeplinită 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 şi tabelul 4.9). Acest calcul se face la etichetele: JMPAM (microadresa 55_H), JMPAI (microadresa 56_H), şi 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 şi respectiv impementat 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 şi 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 $4D_H$), şi respectiv CALLAX (microadresa $4E_H$). Modul de adresare direct este evident ilegal şi la instrucțiunea CALL. Acest lucru a fost justificat la 4.1 2.C şi respectiv impementat în microcod la eticheta CALLAD (microadresa $4B_H$).

În clasa D regăsim instrucțiunile fără operanzi (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** şi cele 20 (în realitate 12!) microrutine **EX_D** este implementată prin două salturi indexate succesive. Primul salt este JUMPI A+(CL_1, CL_0) şi 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+(CL_1, CL_0) realizează ramificația pe cele 4 direcții majore: A, B, C, D (cele 4 clase de instrucțiuni). Dacă instrucțiunea curentă este din clasa D, atunci CL_1, CL_0=1,1 şi succesorul JUMPI A+(CL_1, CL_0) 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+(IR_{11}÷IR_8, 0), care implementează legătura spre cele 20 (în realitate 12!) microrutine **EX_D** (vezi fig 4.35 şi microprogramul de emulare din tabelul 4.9).

Deşi 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 şi CCC) au alocată o microrutină de execuție comună (vezi prima observație de la 4.1.4 şi microprogramul de emulare din tabelul 4.9).

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

Utilizarea celor două măşti 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 şi desigur o reducere cantitativă a *hardware*-ului de implementare şi a complexității acestuia.

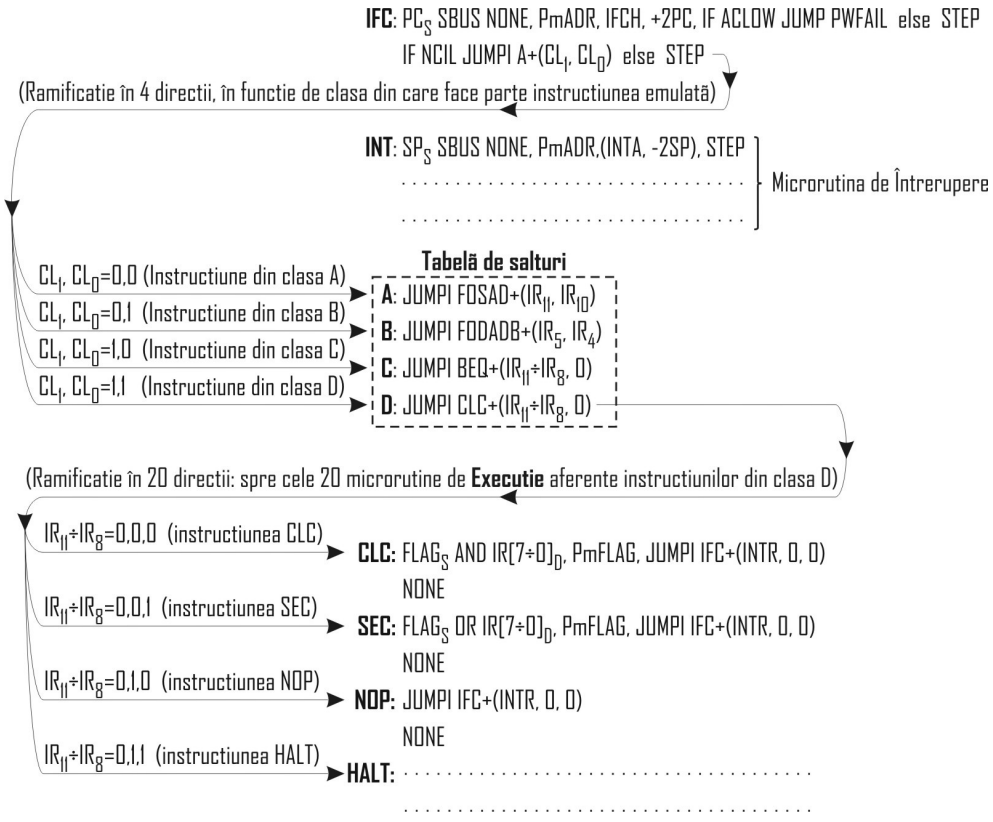


Fig 4.35 Mecanismul de selecție a microrutinelor în cadrul procesului de emulare a unei instrucțiuni din clasa D

După cum am menționat deja la instrucțiunile din clasa A, toate microrutinele 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).