CUPRINS

CUPRINS **1**
1. INTRODUCERE ÎN SISTEMELE DE PROCESARE A INFORMAȚIEI **4**
1.1. CALCULATORUL – DISPOZITIV UNIVERSAL DE CALCUL 8
1.2. STRUCTURA UNUI SISTEM DE CALCUL. PRINCIPII DE BAZĂ 15
1.3. EXERCIȚII ŞI PROBLEME 23
2. REPREZENTAREA INFORMAȚIILOR. CONVERSII DE VALORI ÎNTRE SISTEME DE NUMERAȚIE. ARITMETICĂ BINARĂ. OPERAȚII LOGICE **25**
2.1. REPREZENTAREA INFORMAȚIILOR ÎN SISTEMELE DE CALCUL 25
2.1.1. REPREZENTAREA INFORMAȚIILOR NUMERICE FĂRĂ SEMN. SISTEME DE NUMERAȚIE 29
2.1.2. REPREZENTAREA NUMERELOR CU SEMN 33
2.1.3. REPREZENTAREA NUMERELOR ÎN VIRGULĂ FLOTANTĂ. FORMATUL IEEE 754 36
2.1.4. REPREZENTAREA INFORMAȚIILOR ALFANUMERICE – STANDARDUL ASCII 40
2.1.5. REPREZENTAREA INFORMAȚIILOR MULTIMEDIA 41
2.2. CODIFICAREA INFORMAȚIEI 43
2.2.1. CODURI ZECIMALE 43
2.2.2. CODURI PENTRU DETECTAREA ŞI CORECTAREA ERORILOR 44
2.3. OPERAȚII LOGICE PE BIȚI. FUNCȚII LOGICE. 45
2.3.1. FUNCȚIA ŞI LOGIC (*AND*) 45
2.3.2. FUNCȚIA SAU LOGIC (*OR*) 46
2.3.3. FUNCȚIA NOT LOGIC 46
2.3.4. FUNCȚIA SAU EXCLUSIV (*XOR*) 46
2.4. EXERCIȚII ŞI PROBLEME 47
3. STRUCTURI LOGICE DIGITALE. TRANZISTORI. PORȚI LOGICE. STRUCTURI LOGICE COMBINAȚIONALE. UNITATEA ARITMETICO-LOGICĂ **50**
3.1. TRANZISTORUL 50
3.2. PORȚI LOGICE 53
3.2.1. INVERSORUL CMOS 53
3.2.2. POARTA NOR (*NOT OR*) 53
3.2.3. POARTA NAND (*NOT AND*) 54
3.3. STRUCTURI LOGICE DIGITALE 56
3.3.1. STRUCTURI LOGICE (CIRCUITE) COMBINAȚIONALE 56
3.3.2. PROIECTAREA UNEI UNITĂȚI DE CALCUL ARITMETICO-LOGIC 65
3.4. EXERCIȚII ŞI PROBLEME 70
4. ELEMENTE PRIMARE DE MEMORARE. REGIŞTRII. MEMORII. AUTOMATE CU NUMĂR FINIT DE STĂRI **78**
4.1. STRUCTURI LOGICE SECVENȚIALE 78
4.1.1. BISTABILUL R-S 79
4.1.2. BISTABILUL DE TIP D (DELAY). 81
4.1.3. REGIŞTRII. 82

4.1.4. MEMORIA 85
4.2. AUTOMATE SECVENȚIALE ŞI PROGRAMABILE 92
4.3. EXERCIȚII ŞI PROBLEME 100
5. MODELUL ARHITECTURAL *VON NEUMANN*. COMPONENTE DE BAZĂ. PRINCIPIILE PROCESĂRII INSTRUCȚIUNILOR 104
5.1. COMPONENTELE DE BAZĂ ALE MODELULUI ARHITECTURAL VON NEUMANN 104
5.2. PRINCIPIILE PROCESĂRII INSTRUCȚIUNILOR 108
5.2.1. CICLUL INSTRUCȚIUNII 109
5.2.2. TIPURI DE INSTRUCȚIUNI 112
5.2.3. CEASUL PROCESORULUI 114
5.3. EXERCIȚII ŞI PROBLEME 115
6. LC-3 – ARHITECTURA SETULUI DE INSTRUCȚIUNI. CALEA FLUXULUI DE DATE. ORGANIZAREA MEMORIEI LA LC-3 [Patt03]. 119
6.1. LC-3 – ARHITECTURA SETULUI DE INSTRUCȚIUNI 119
6.1.1. LC-3 ISA: ORGANIZAREA MEMORIEI ŞI SETUL DE REGIŞTII GENERALI 120
6.1.2. LC-3 ISA: FORMATUL INSTRUCȚIUNII ŞI SETUL DE INSTRUCȚIUNI 121
6.1.3. LC-3 ISA: APLICAȚII REZOLVATE 141
6.2. FLUXUL DE DATE DA LC-3 ISA 145
6.3. EXERCIȚII ŞI PROBLEME 148
7. LIMBAJUL DE ASAMBLARE AFERENT ARHITECTURII LC-3. ASAMBLORUL. ETAPELE GENERĂRII CODULUI MAŞINĂ. TABELA DE SIMBOLURI 156
7.1. MOTIVE PENTRU A PROGRAMA ÎN LIMBAJ DE ASAMBLARE 156
7.2. SINTAXA ASAMBLOR LC-3 [Patt03] 158
7.2.1. CORESPONDENȚA LIMBAJ DE ASAMBLARE LC-3 – LIMBAJ MAŞINĂ SPECIFIC LC-3 ISA 162
7.2.2. APELURI SISTEM 163
7.3. PROCESUL DE ASAMBLARE – ETAPELE GENERĂRII CODULUI MAŞINĂ 164
7.3.1. GENERAREA TABELEI DE SIMBOLURI (TS) 165
7.3.2. OBȚINEREA CODULUI ÎN LIMBAJUL MAŞINĂ 166
7.3.3. CREAREA LEGĂTURILOR ÎNTRE MODULE, ÎNCĂRCAREA ŞI LANSAREA ÎN EXECUȚIE 169
7.4. EXERCIȚII ŞI PROBLEME 170
8. ÎNTRERUPERI SOFTWARE LA NIVEL *LOW*. APELURI DE SUBRUTINE – DIRECTE ŞI INDIRECTE. REVENIRI. SALVAREA ŞI RESTAURAREA REGIŞTRIILOR. STRATEGIILE "*CALLER-SAVE*" RESPECTIV "*CALLEE SAVE*" 177
8.1. ÎNTRERUPERI. DEFINIȚIE. CLASIFICARE 177
8.2. ÎNTRERUPERI SOFTWARE LA NIVEL *LOW* – INSTRUCȚIUNILE *TRAP* 178
8.2.1. RUTINELE DE TRATARE AFERENTE APELURILOR SISTEM LA LC-3 ISA 179
8.2.2. SALVAREA ŞI RESTAUREA REGIŞTRILOR ÎN CAZUL RUTINELOR DE SERVICIU 190
8.3. SUBRUTINE 192
8.3.1. MECANISMUL DE APEL SUBRUTINE ŞI REVENIRE 193
8.3.2. TRANSFERUL PARAMETRILOR CĂTRE ŞI DE LA SUBRUTINE 198
8.3.3. SALVAREA ŞI RESTAUREA REGIŞTRILOR ÎN CAZUL SUBRUTINELOR. RUTINE (FUNCȚII) DE BIBLIOTECĂ 199
8.4. EXERCIȚII ŞI PROBLEME 200

9. STIVA – STRUCTURĂ. PRINCIPIU DE FUNCȚIONARE. OPERAȚII AFERENTE (*PUSH & POP*) 208
9.1. STRUCTURA DE DATE DE TIP STIVĂ ____ 208
9.1.1. PRINCIPIU DE FUNCȚIONARE ____ 208
9.1.2. IMPLEMENTARE HARDWARE ŞI SOFTWARE. ____ 208
9.1.3. OPERAȚII AFERENTE (PUSH & POP) ____ 211
9.1.4. MODUL DE LUCRU PRIN ÎNTRERUPERI HARDWARE. ____ 214
9.2. OPERAȚII ARITMETICE FOLOSIND STIVA ____ 222
9.3. CONVERSIA INFORMAȚIEI DIN FORMAT ASCII ÎN ZECIMAL ____ 225
9.3.1. CONVERSIA DIN ASCII ÎN BINAR ____ 226
9.3.2. CONVERSIA DIN ZECIMAL ÎN ASCII ____ 230
9.4. EXERCIȚII ŞI PROBLEME ____ 233
10. FACILITĂȚI ALE MAŞINII PENTRU IMPLEMENTAREA ÎN HARDWARE A FUNCȚIILOR DIN PROGRAMELE DE NIVEL ÎNALT. STIVA DE DATE AFERENTĂ FUNCȚIILOR 237
10.1. OBȚINEREA CODULUI OBIECT PENTRU O ARHITECTURĂ DATĂ ___ 237
10.1.1. DESCRIEREA COMPONENTELOR UNUI COMPILATOR ____ 238
10.1.2. ETAPELE PARCURSE PENTRU RECOMPILAREA INSTRUMENTELOR *SIMPLESCALAR 3.0* [Flo05] ____ 246
10.2. IMPLEMENTAREA GESTIUNII STIVELOR DE DATE ASOCIATE FUNCȚIILOR C ____ 249
10.2.1. SUBPROGRAME. GENERALITĂȚI. FUNCȚII C ____ 249
10.2.2. STIVA DE DATE AFERENTĂ FUNCȚIILOR [Patt03, Vin03] ____ 256
10.3. EXERCIȚII ŞI PROBLEME ____ 260
11. INTRODUCERE ÎN *RECURSIVITATE*. COMPARAȚIA DINTRE *RECURSIV* ŞI *ITERATIV* ÎN ALEGEREA ALGORITMULUI DE REZOLVARE A PROBLEMELOR. AVANTAJE / DEZAVANTAJE 265
11.1. SCOP ŞI COMPETENȚE NECESARE ____ 265
11.2. RECURSIVITATEA ____ 265
11.2.1. STIVA DE DATE ASOCIATĂ UNEI FUNCȚII [Patt03] ____ 266
11.3. IMPLEMENTAREA RECURSIVITĂȚII LA NIVELUL STIVEI DE DATE ____ 274
11.4. TIPURI DE FUNCȚII RECURSIVE. ELIMINAREA RECURSIVITĂȚII ___ 277
11.5. EXERCIȚII ŞI PROBLEME ____ 278
12. POINTERI ŞI TABLOURI. TRANSFERUL PARAMETRILOR PRIN REFERINȚĂ. POINTERI SPRE FUNCȚII 282
12.1. INTRODUCERE ____ 282
12.2. SEMNIFICAȚIE ŞI DECLARARE ____ 284
12.3. ALOCAREA ŞI ACCESAREA DE VARIABILE. LEGĂTURA DINTRE NIVELUL HIGH ŞI LOW VĂZUTĂ PRIN INTERMEDIUL MODURILOR DE ADRESARE ____ 287
12.4. TRANSFERUL PARAMETRILOR PRIN REFERINȚĂ LA APELUL FUNCȚIILOR ____ 290
12.5. POINTERI SPRE FUNCȚII ____ 295
12.6. TABLOURI. RELAȚIA DINTRE POINTERI ŞI TABLOURI ____ 302
12.6.1. FUNCȚII CU PARAMETRI DE TIP TABLOU (VECTOR) ____ 305
12.7. EXERCIȚII ŞI PROBLEME ____ 307
BIBLIOGRAFIE 311

1. INTRODUCERE ÎN SISTEMELE DE PROCESARE A INFORMAȚIEI

Ştiința Calculatoarelor constituie unul din cele mai **dinamice** domenii ale ştiinţelor inginereşti. Dezvoltarea fără precedent a domeniului *Calculatoare şi tehnologia informației* ca o componentă fundamentală a societății actuale (*societate informațională*), aflată într-o continuă schimbare şi progres, este evidentă şi face parte integrantă din realitatea pe care o trăim. Evoluția explozivă a tehnologiilor de înaltă performanță, coroborată cu acumulări teoretice şi experimentale, au făcut posibilă pătrunderea în cele mai diverse domenii ale vieții sociale, economice, industriale, culturale, a ceea ce poartă în mod generic numele de "*sistem de calcul / calculator, tehnică / echipament de calcul*". Importanța informației şi a sistemelor de comunicații cu infrastructura aferentă pentru societate şi economie se accentuează odată cu valoarea şi cantitatea informației transmisă şi stocată pe aceste sisteme. În acest context apare în mod stringent necesitatea îmbunătățirii performanțelor sistemelor de calcul actuale atât din punct de vedere cantitativ cât mai ales din punct de vedere calitativ. De asemenea, prezentul tehnologiei informației, ca să nu mai spunem de viitor, centrat pe Internet şi tehnologia *World Wide Web*, impun ca alături de performanța în sine, fiabilitatea, disponibilitatea şi scalabilitatea să devină criterii esențiale, ceea ce implică iarăşi necesitatea unei noi viziuni pentru proiectantul de sisteme de calcul. Criticalitatea în societatea informațională, unde informația circulă prin cyberspații[1] fără constrângeri de distanță sau viteză, se datorează în egală măsură dependenței sporite de informație, sistemelor care o furnizează dar şi vulnerabilității crescânde pe fondul unui spectru larg de amenințări de genul războiului şi poluării informaționale (război cibernetic, etc.).

Într-adevăr, calculatoarele au făcut progrese importante, iar microprocesoarele sunt principalele răspunzătoare pentru această creştere deosebită a performanței. Performanța microprocesoarelor a evoluat în

[1] Conform enciclopediei Wikipedia [Wik], termenul de „cyberspațiu" (Cyberspace) reprezintă o metaforă atribuită rețelelor de sisteme electronice de calcul care permit stocarea de date şi comunicarea *online*. Deşi în anumite lucrări, „cyberspațiul" este folosit cu sensul de Internet, termenul trebuie înțeles ca şi un spațiu care cuprinde, identitățile şi obiectele care există în rețelele de calculatoare şi sunt folosite de indivizii umani în diverse scopuri [Chi04].

ultimii 10 ani într-un ritm exponențial, cu o rată aproximativă de 60% pe an. Cercetătorii susțin că progresele arhitecturale au o pondere mai mare decât cele tehnologice. Tendințele tehnologice se referă la creşterea gradului de integrare al tranzistorilor pe cip, creşterea frecvenței ceasului procesorului, diminuarea timpul de acces la memorie, reducerea costurilor de implementare hardware la aceeaşi putere de calcul ori capacitate de memorare etc. Tendințele arhitecturale urmăresc exploatarea şi creşterea paralelismului la nivelul instrucțiunilor atât prin tehnici statice cât şi dinamice sau hibride (cazul arhitecturii IA-64, procesorul *Intel Itanium*), o ierarhizare a sistemului de memorie prin utilizarea unor arhitecturi evoluate de memorii tip cache, reducerea latentei căii critice de program, etc.

În cadrul unor prestigioase conferințe ştiințifice internaționale dedicate microarhitecturilor avansate de procesare a informației (1999-2007) se agreează tot mai mult ideea potrivit căreia pentru a continua şi în viitor creşterea exponențială a performanței microprocesoarelor, sunt necesare concepte noi, revoluționare chiar, pentru depăşirea limitărilor paradigmei actuale din punct de vedere conceptual. Există încă o puternică tendință de specializare îngustă care face adesea ca abordarea domeniului să fie una închisă în tipare preconcepute. Abordări recente, arată însă că sinergia unor instrumente aparent disjuncte ale ştiinței calculatoarelor converge spre realizări novatoare ale acestui domeniu de cercetare (arhitectura calculatoarelor şi inteligența artificială). Alte posibile soluții constau în abordări integratoare de gen *hardware-software*, *tehnologie-arhitectură*, *algoritmi*, *concepte*, *metode*.

Cercetările în domeniul Ştiinței Calculatoarelor trebuie să aibă un **caracter fundamental**, şi se recomandă a se efectua în universități întrucât implică investigarea unor aspecte noi, neimplementate în actualele microprocesoare. Comunitatea academică şi industrială internațională investeşte bugete importante în cercetarea arhitecturilor noi de calcul. Astfel de exemplu, programe de cercetare europene de tip FP6 (*HiPEAC - High-Performance Embedded Architecture and Compilation*) sunt finanțate de către Uniunea Europeana cu fonduri de milioane de euro. Abordarea cantitativă şi calitativă a acestui domeniu de studiu şi cercetare în România reprezintă o necesitate formativă dar şi o necesitate legată de **menținerea cercetării ştiințifice românești în cadrul tendințelor şi standardelor actuale de calitate** din Europa de Vest, Japonia şi SUA. Consider că mediul universitar este ideal pentru dezvoltarea unor asemenea cercetări fundamentale cu privire la tehnologia informației, inclusiv prin diseminarea rezultatelor către studenții anilor terminali şi celor masteranzi.

Această lucrare reprezintă o introducere axată pe **înțelegerea principiilor fundamentale ale ştiinței şi ingineriei calculatoarelor** şi este destinată în primul studenților din anul I ai studiilor de licență, dar şi celor care doresc să pătrundă în tainele calculatoarelor şi care văd în acestea nu doar simple instrumente de lucru. Cartea de față, bogată în aplicații practice, se bazează în parte pe structura lucrărilor *Introduction to Computing Systems: from bits & gates to C & beyond* [Patt03] şi *Computer Organisation and Design*: *The Hardware/Software Interface* [Pat05], cărți folosite ca referință la cursul de *Introducting to Computing Systems* introdus pentru prima dată în universitățile americane în toamna anului **1995** de către Profesorul Kevin Compton de la Universitatea Michigan. În acest moment este predat în mai bine de 75 de universități americane şi la cursuri înrudite de gen: *Introduction to Computing for Biomedical Engineers*, *Computer Systems and Programming in C,* abordarea optimă fiind de 2 semestre. De asemenea, mai sunt folosite drept referințe o suită de peste 30 de lucrări – cărți şi articole ştiințifice – atât din domeniul software (centrate pe limbaje şi tehnici de programare, compilatoare, structuri de date) cât şi hardware (focalizate pe circuite digitale, automate secvențiale, microarhitecturi de procesare) cu scopul de a exemplifica mai bine necesitatea abordării integratoare *hardware-software* a domeniului ştiinței şi ingineriei calculatoarelor. Accentul în lucrarea de față nu este pus pe memorarea detaliilor tehnice ci pe **interfața dintre ceea ce software-ul necesită şi respectiv ceea ce hardware-ul poate executa**. Sunt prezentate noțiuni introductive dar cu caracter fundamental aferente mai multor discipline din domeniul ştiinței calculatoarelor: limbaje de programare, algoritmi şi structuri de date, compilatoare, sisteme de operare, respectiv – arhitectura calculatoarelor, sisteme cu microprocesoare.

❑ Abordarea ***bottom-up*** a cărții este axată pe două componente:

- **Structura de bază a calculatoarelor** folosind simulatorul LC-3 (*Little Computer* – www.ece.utexas.edu/~ambler/ee306/Software&Doc/**LC3**WinGuide.pdf) dezvoltat la Catedra de Calculatoare şi Inginerie Electrică, Universitatea din Austin, Texas. Se pornește de la tranzistoare MOS, porți logice, bistabili, structuri logice (Multiplexoare, Decodificatoare, Sumatoare, memorii). Se continuă cu modelul de execuție ***von Neumann***, cu arhitectura unui calculator (procesor) simplu LC-3, cu limbajul de asamblare aferent acestui procesor. Sunt studiate de asemenea: interfața cu tastatura şi monitorul, apelurile sistem (întreruperi software reprezentând servicii ale sistemului de operare prin instrucțiuni dedicate), modurile de adresare pentru instrucțiunile

cu referire la memorie (load / store), mecanismul de apel şi revenire din subrutină.

- **Exemplificarea la nivelul limbajului C şi legătura sau implicațiile în hardware a structurilor de control, a diverselor tehnici de alocare a memoriei, lucrul cu stiva, apel de funcții şi revenire, stivele de date aferente funcțiilor, transmiterea parametrilor în cazul apelului, recursivitate (**care pare „*magică*", dincolo de înțelegere**),** tablouri şi chiar structuri elementare de date. Se va insista, de asemenea, pe lucrul cu pointeri; pe modul în care C-ul (limbajul, compilatorul) alocă şi accesează variabile. Se va exemplifica arătându-se diferența dintre adresa unei locații de memorie (*p* din declarația int *p;) şi conținutul acesteia (*p – din aceeaşi declarație). Se va demonstra ineficiența la nivelul arhitecturii a modului de adresare indirect memorie (la o adresă în memorie se află adresa operandului care va fi folosit – **cap, în cazul transmiterii de parametrii unei funcții prin referință).

Aspectul novator al abordării va consta în prezentarea **interfeței arhitectură (ISA, limbaj de asamblare specific) – aplicație de nivel înalt (**incluzând **compilatorul ca şi generator de cod pentru respectiva arhitectură).**

🖬 Ce este LC-3 ?

- Simulator care descrie funcționarea unei arhitecturi pe 16 biți; Arhitectura înglobează cele mai importante caracteristici ale procesoarelor Intel 8088 – folosit în primul sistem IBM PC în 1981, Motorola 68000 – folosit la Macintosh 1984, sau Intel Pentium III integrat în sistemele anilor 2000. Simulatorul permite execuția programelor scrise în limbaj de asamblare LC-3, depanare interactivă – stabilire de puncte de întrerupere, execuție pas cu pas, vizualizare conținut regiştrii. În urma etapei de asamblare codul sursă asamblare este translatat în limbaj maşină. De asemenea, poate fi vizualizat şi fişierul care conține tabela de simboluri.

- Include porturi de I/O, interfațare cu tastatura şi monitorul, apeluri sistem (întreruperi software reprezentând servicii ale sistemului de operare prin instrucțiuni), salturi condiționate pe coduri de condiție =0, ≠0, set minimal de instrucțiuni, moduri de adresare pentru instrucțiunile cu referire la memorie (load / store) – indirect registru, indexat, mecanism de apel şi revenire din subrutină (recursivitatea determină necesitarea salvării adresei de revenire în programul apelant în momentul apelului).

- Simulatorul arhitecturii LC-3 permite testare / depanare a programelor scrise în limbaj de asamblare, stabilire de puncte de întrerupere. De la adresa http://users.ece.utexas.edu/~ambler/ee306/software&doc.htm pot fi descărcate în mod gratuit pentru utilizare în scop didactic atât simulatorul LC-3 (LC301.exe), un simulator realizat în Macromedia Flash Player versiunea 6, care descrie operațiile de citire – scriere dintr-o memorie cu 4 locații având 3 biți fiecare (memory.exe), precum şi ghidul de utilizare al simulatorului LC-3.

1.1. CALCULATORUL – DISPOZITIV UNIVERSAL DE CALCUL

Universalitatea

Un sistem de calcul[2] (calculatorul) trebuie să ofere următoarele funcționalități:

- cum să adune / scadă două numere
- cum să înmulțească / împartă două numere
- cum să ordoneze alfabetic
- nu trebuie proiectat sau cumpărat un calculator nou pentru fiecare nouă operație care trebuie rezolvată.
- dacă este definit un alt set de instrucțiuni acelaşi calculator trebuie să opereze cu ele.

Istoric

Exceptând *abacul* (strămoşul socotitoarei – folosit pentru prima dată, cu peste 2000 de ani în urmă în China), se poate spune că maşinile de calcul au apărut cu mult (≈300 de ani) înaintea calculatoarelor de azi. Acestea pot fi clasificate în:

- Dispozitive **analogice** (măsurarea unor cantități fizice cum ar fi tensiune, curent). Maşini de adunat, înmulțit folosind diverse rigle

[2] Conform enciclopediei Wikipedia [Wik], termenul de **sistem de calcul** reprezintă o combinație de componente hardware şi software. Un sistem de calcul tipic conține o memorie şi un set de stări care definesc legătura dintre intrările şi ieşirile sistemului. Pe parcursul acestei lucrări s-a folosit pentru simplitate în loc de sistem de calcul noțiunea de calculator. Pe lângă calculator însă, sisteme de calcul mai pot fi: sisteme dedicate din aparatele electrocasnice, din automobile, dispozitive PDA (*Personal digital assistants*), etc.

gradate logaritmice. Sunt caracterizate de **acuratețe scăzută** (Ex: ceasul).

- Dispozitive **digitale** (calculează manipulând un set finit de digiți sau caractere).

- **Prima maşină de calcul** funcțională a fost construită de către **Blaise Pascal** în anul **1642**. Reprezenta o socotitoare mecanică, construită din roți dințate şi o manivelă, cu care se puteau efectua *adunări* şi *scăderi*.
- În anul **1674**, **Gottfried von Leibniz** a construit o maşină de calcul care pe lângă adunări şi scăderi efectua şi *înmulțiri* şi *împărțiri*. Căutând să simplifice mecanismele de efectuare a calculelor a considerat cel mai potrivit *sistemul binar de numerație*.
- **Charles Babbage** a proiectat în **1840** prima maşină analitică de uz general, care (dacă ar fi fost posibilă construirea ei) ar fi fost programabilă, având patru componente principale: *magazia* (memoria), *moara* (unitatea de calcul), *secțiunea de intrare* (cititorul de cartele) şi *secțiunea de ieşire* (perforatorul sau imprimanta).
- În **1854** matematicianul englez **George Boole** a inventat **calculul logic** care foloseşte numai două valori (*adevărat* şi *fals*) cu care se pot efectua operații de tip And, Or, Not. Aceste funcții (operatori), numite ulterior **algebră booleană**, pot fi uşor simulate cu ajutorul unei **rețele de comutatoare**.
- **Konrad Zuse** (student german) a creat la sfârşitul anilor **1930** o serie de maşini de calcul folosind **relee electromagnetice**.
- **Calculatoarele electronice** au fost inițial propuse sub forma unui aparat abstract (**descriere matematică**) care poate realiza sau simula orice tip de maşină mecanică. Forma teoretică a fost introdusă de **Alan Turing** în **1936** la Universitatea Cambridge sub denumirea de **maşina Turing universală**.

Primele calculatoare construite erau foarte diferite de cele de astăzi. Nu doar ca dimensiuni şi capacitate (care depind doar de tehnologie), ci referitor la structura lor fundamentală. Acele calculatoare erau construite pentru a rezolva o *singură problemă*; nu erau universale. Ele constau dintr-o colecție de unități funcționale, care puteau face calcule simple. „Programatorii" aveau sarcina de a conecta unitățile funcționale între ele cu fire (sârme), pe care le inserau manual în tot felul de mufe. De exemplu, dacă vroiau să calculeze $(a+b)^2$, programatorii luau o unitate care făcea adunări şi una care făcea înmulțiri şi le cuplau ca în figura 1.1.

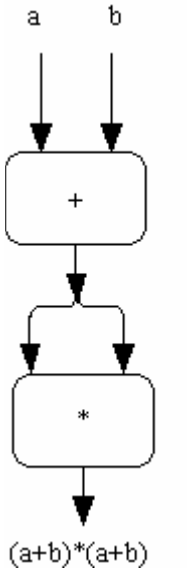


Figura 1.1. Implementarea în hardware a expresiei $(a+b)^2$. Variabilele devin sârme, iar operaţiile sunt executate de unităţi funcţionale

Ideea de a descrie un program folosind un limbaj (şi nu prin conexiuni între unităţi funcţionale) este mai veche; în 1936 Alan Turing folosise noţiunea de „**maşină Turing universală**" (**U**) pentru a descrie un calculator universal, care poate executa orice program. Programele erau stocate în memoria calculatorului, reprezentate ca şiruri de numere. Maşina Turing (**T**) este un calculator abstractizat (figurahttp://www.cs.cmu.edu/~mihaib/articles/complex/complex-html.html - turing 1.2) compusă din următoarele piese [Bud99]:

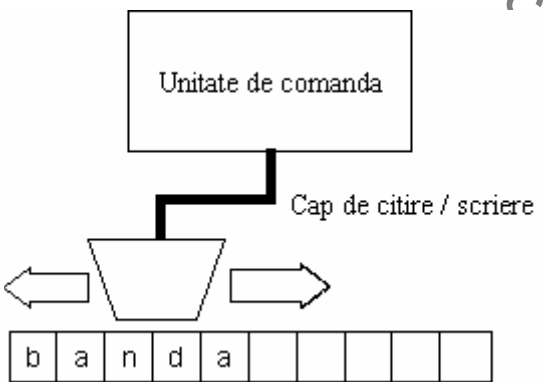


Figura 1.2. Maşina *Turing*

1. O bandă infinită de hârtie cu pătrățele; în fiecare pătrățel se poate scrie exact un caracter din alfabetul nostru; banda este inițial plină cu „spații”, mai puțin partea de la început, unde este scris şirul cu datele de intrare –memorie cu acces linear;
2. Un cap de citire-scriere, care se poate mişca deasupra benzii, la stânga sau la dreapta;
3. O unitate de control, care conține un număr finit de reguli. Unitatea de control este la fiecare moment dat într-o *stare*; stările posibile sunt fixate dinainte, şi sunt în număr finit. Fiecare regulă are forma următoare:

Dacă
- sunt în starea Q1;
- sub capul de citire este litera X;

atunci:
- trec în starea Q2;
- scriu pe bandă litera Y;
- mut capul de citire/scriere în direcția D.

În pofida simplității ei, mașina Turing poate calcula orice poate fi calculat cu cele mai performante supercomputere. Un algoritm de calcul este descris de o astfel de mașină, prin toate stările posibile, şi toate aceste reguli, numite *reguli de tranziție*, care indică cum se trece de la o stare la alta.

De exemplu:

```
if ((stare_veche==Qi)&&(intrare==Ti))
{
        ieşire=Oj;
        stare_nouă=Qj;
}
```

Orice alte modele de calcul care au fost propuse de-a lungul timpului, au fost dovedite a fi mai puțin expresive, sau tot atât de expresive cât mașina Turing. Nimeni nu a fost în stare, până în prezent, să demonstreze că „mașina Turing are limitări”: adică, dispunând de operații elementare capabile să exprime orice algoritm, să ofere ceva care să fie construibil, şi să poată face lucruri pe care mașina Turing nu le poate face.

Din cauza asta logicianul Alonzo Church a emis ipoteza că **mașina Turing este modelul cel mai general de calcul care poate fi propus**; acest enunț, care nu este demonstrabil în sens matematic, se numeşte „**Teza lui Church**”. Acesta este un **postulat** asupra căruia trebuie căzut de acord înainte de a putea analiza orice alt lucru privitor la teoria complexității.

Complexitatea unui model de calcul prin prisma modelului Turing, se defineşte prin:

Timpul de calcul – pentru un şir dat la intrare, este numărul de mutări făcut de maşina Turing înainte de a intra în starea „*terminat*";

Spaţiul – consumat pentru un şir de intrare, este numărul de căsuţe de pe bandă pe care algoritmul le foloseşte în timpul execuţiei sale.

Anticipând puţin complexitatea unui algoritm este dată de timpul de execuţie al acestuia în situaţia cea mai defavorabilă. *Timpul de execuţie al unui algoritm* pentru un anumit set de date de intrare este determinat de numărul de operaţii primitive sau "*paşi*" executaţi şi se calculează ca suma tuturor timpilor de execuţie corespunzători (pas = linia **i** din pseudocod executată într-o durată (presupusă) constantă de timp **ci**).

Când se spune că *maşina Turing este la fel de puternică ca orice alt model de calcul*, nu înseamnă că poate calcula la fel de repede ca orice alt model de calcul, ci că poate calcula aceleaşi lucruri.

Se pot imagina tot felul de modificări minore ale maşinii Turing, care o vor face să poată rezolva anumite probleme mai repede. De exemplu, putem să ne imaginăm că maşina are dreptul să mute capul la orice căsuţă dintr-o singură mişcare, fără să aibă nevoie să meargă pas-cu-pas; atunci banda s-ar comporta mai asemănător cu o memorie RAM obişnuită.

Simulări; maşina Turing universală [Bud99]

Demonstraţia faptului că maşina Turing este atât de puternică încât este echivalentă cu orice alt model propus de calcul s-a făcut prin simulare. De fapt maşina Turing este atât de puternică încât se poate construi o maşină Turing care să simuleze orice altă maşina Turing posibilă. Atât calculatoarele cât şi maşina universală Turing pot calcula orice întrucât ele sunt **programabile**.

Pentru a înţelege cum este posibil aşa ceva, trebuie realizate două lucruri:

- Descrierea oricărei maşini Turing (**T**) este finită, şi poate fi făcută cu alfabetul nostru. Dacă am o maşină Turing, pot enumera stările ei şi regulile de tranziţie sub forma unui şir de caractere;
- Maşina universală (**U**) primeşte două intrări pe bandă: una este descrierea maşinii de simulat (ce trebuie să calculeze), iar a doua este intrarea pentru care trebuie să simuleze maşina. Maşina universală apoi urmăreşte regulile de tranziţie ale maşinii simulate, folosind propria ei bandă în acest scop.

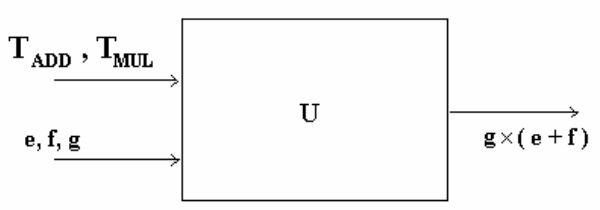


Figura 1.3. Maşina *Turing* universală

(model abstract de calcul al unui produs dintre un termen şi o sumă)

- Dezvoltarea dispozitivelor de calcul a fost influențată de cel de-al doilea război mondial prin încercările de calcul a traiectoriilor rachetelor Collosus şi Eniac şi respectiv de decodare a codului Enigma folosit de Germania pentru protecția mesajelor. **Primul calculator digital de succes**, **ENIAC** (*Electronic Numeric Integrator and Computer*) înlocuieşte releele electromagnetice cu tuburile electronice. A fost construit în **1946** de **John Mauchly** şi **Presper Eckert** de la Universitatea Pensylvania şi conținea 18000 de tuburi electronice şi 1500 de relee. Cântărea 20 de tone, avea 20 de regiştri, care reținea fiecare câte un număr zecimal de 10 cifre, era programat prin intermediul a 6000 de comutatoare şi era posibilă efectuarea a până la 500 de adunări sau scăderi şi 300 de înmulțiri pe secundă. A fost folosit de armata americană până în **1952**.
- În anii '40, matematicianul **John von Neumann** analizează starea de fapt a calculatoarelor şi scrie în **1945** un raport intitulat „First Draft of a Report on the EDVAC" (Prima ciornă a unui raport despre **EDVAC – Electronic Discrete Variable Automatic Computer**), în care sugerează o arhitectura revoluționară. În această arhitectură, programul nu mai este reprezentat de felul în care sunt cuplate unitățile funcționale, ci este stocat în memorie, fiind descris folosind un limbaj numit cod-maşină. În cod-maşină, operațiile de executat sunt codificate sub forma unor numere numite *instrucțiuni*. Programul de executat este descris printr-un şir de instrucțiuni, care se execută consecutiv. Pe lângă unitățile funcționale care fac operații aritmetice, calculatorul mai are o unitate de control, care *citeşte secvențial instrucțiunile programului şi care trimite semnale între unitățile funcționale pentru a executa aceste instrucțiuni*. *Rezultatele intermediare sunt stocate în memorie*. Această arhitectură se numeşte „**von Neumann**".

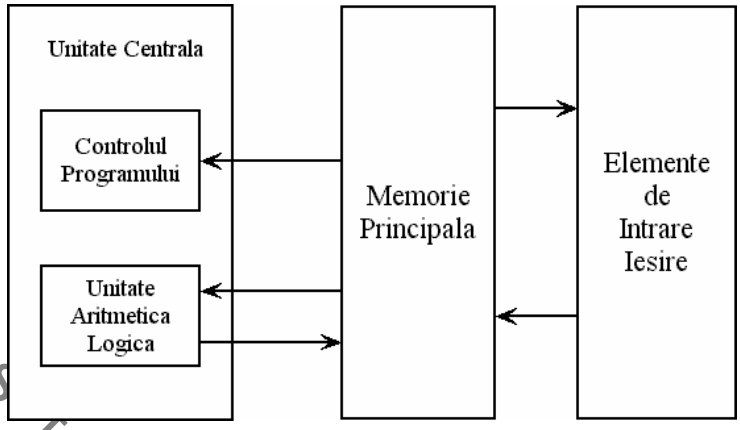


Figura 1.4. Schema simplificată a arhitecturii *von Neumann*

- Marea majoritate a calculatoarelor din ziua de azi sunt bazate pe această arhitectură; noţiunea de limbaj-maşină, şi cea înrudită, de limbaj de programare, folosite pentru descrierea programelor, sunt concepte foarte naturale pentru toţi cei care manipulează calculatoarele. Von Neumann propune **calculatorul** să fie văzut ca sistem de procesare a informaţiei (bazat pe procesor – CPU), adică un **mecanism care direcţionează dar şi realizează procesarea informaţiei**.
- Apariţia **tranzistorului,** înlocuitorul tuburilor electronice – a generat începutul celei de-**a doua generaţii de calculatoare electronice, a minicalculatoarelor (1955-1965). PDP-1**, realizat de firma Digital Equipment Corporation este un reprezentant al acestor calculatoare. Faţă de cel mai rapid calculator din lume la acea dată, IBM 7090, un calculator cu tranzistori, PDP-1 era de 2 ori mai rapid si de 9 ori mai ieftin.
- **A treia generaţie (1965-1980)** – a debutat prin inventarea de către **Robert Noyce a circuitului integrat de siliciu** care permitea montarea pe o singură pastilă de siliciu numită **cip**, de câţiva centimetri pătraţi, a zeci, apoi mii de tranzistoare. Descoperirea cipului a permis constructorilor de calculatoare să creeze maşini mai mici, mai rapide şi mai ieftine decât cele precedente. A apărut noţiunea de *familie de calculatoare*: maşini care au acelaşi limbaj de asamblare dar au puteri (performanţe) şi capacităţi diferite. S-a introdus noţiunea de *multitasking*; aceasta implică existenţa în memorie a mai multor programe în acelaşi timp: când unul dintre ele aşteaptă terminarea unei operaţii de intrare / ieşire, un altul poate efectua calcule.
- **A patra generaţie de calculatoare (**începând cu **1980)** – poartă amprenta **circuitelor integrate pe scară largă – VLSI**. Această tehnologie a permis a permis plasarea pe un singur cip a sute de

milioane de componente electronice elementare. Datorită prețului scăzut al calculatoarelor s-a creat posibilitatea achiziționării acestora de fiecare persoană; a început perioada calculatoarelor personale (*PC*). Procesoarele anilor ’80 erau construite din 10 sau mai multe plăci electronice, fiecare de dimensiuni ≈18” (inch) şi conținând 50 sau mai multe componente electronice integrate. La nivelul anilor 2000 procesoarele sunt integrate pe o singură pastilă de siliciu de dimensiune sub 1” (inch). În cazul microprocesoarelor, gradul de integrare al tranzistorilor pe cip creşte cu cca. 55% pe an. Tehnologia de integrare a microprocesoarelor a evoluat de la 10 microni (1971) la 0.18 microni (2001), la 0.13 microni (2003) – versiunea Northwood a procesorului Intel Pentium 4 şi 0.09 microni (2004) – versiunea Prescott a aceluiaşi procesor. Frecvența ceasului creşte şi ea cu cca. 50% pe an.

1.2. STRUCTURA UNUI SISTEM DE CALCUL. PRINCIPII DE BAZĂ

***Principiul întâi – P1*:** Toate calculatoarele (atât cel mai mare – resurse foarte mari, cât şi cel mai mic, atât cel mai rapid – frecvență ridicată, cât şi cel mai lent, atât cel mai scump cât şi cel mai ieftin) sunt capabile să facă exact aceleaşi lucruri dacă au **suficient timp** şi **suficientă memorie**.

„Inteligența” aparentă a unui sistem de calcul provine din cantitatea de inteligență umană investită în respectivul program aflat în execuție. Marvin Minski, afirma despre programele de Inteligență Artificială din anii ’80 că: „aceste programe care par inteligente iau decizii la fel de proaste ca şi omul numai că mult mai repede” [Min82].

***Principiul al doilea – P2*:** Toate problemele ce se doresc a fi rezolvate, sistemele de operare, compilatoarele, etc., sunt scrise într-un limbaj (şi într-o limbă – uzual engleza). Dar toate sunt „rezolvate” prin electroni şi caracteristici ale acestora (deplasare, viteză, etc). Practic, de la problema de rezolvat la tensiunea care antrenează (direcționează) fluxul de electroni au loc o serie de transformări sistematice (dezvoltate şi îmbunătățite pe parcursul a mai bine de 50 de ani).

Viitorul va aduce probabil calculatoare cuantice, bazate pe procesoare moleculare organice, conform calculabilității ADN, etc.

Nivele de transformare: de la Problemă de nivel înalt la Circuit electronic

Etapele succesive prin care se ajunge de la problema de rezolvat (descrisă într-un mod abstract) până la tensiunea care antrenează fluxul de electroni sunt următoarele:

1. Problemă (*High level*)
2. Algoritm
3. Limbaj de programare
4. Arhitectura Setului de Instrucţiuni (ISA)
5. Determinarea optimă a *microarhitecturii*
6. Proiectarea la nivel de circuit logic (combinaţional / secvenţial)
7. Implementarea circuitelor folosind tehnologia CMOS

1. **Problemele** sunt definite în limbaj natural. Dezavantaj – *ambiguitatea*.
2. **Algoritmul** – reprezintă o succesiune finită şi ordonată de operaţii univoc determinate, efectuate mecanic, care aplicate datelor iniţiale ale unei probleme dintr-o clasă dată, asigură obţinerea soluţiei acelei probleme. Cu alte cuvinte un algoritm este orice procedură de calcul bine definită care primeşte o anumită valoare sau o mulţime de valori ca date de intrare şi produce o anumită valoare sau mulţime de valori ca date de ieşire. Comportarea unui algoritm poate fi diferită în funcţie de datele de intrare. Proprietăţile algoritmilor sunt:
   - **Claritatea** – operaţiile algoritmului şi succesiunea executării lor trebuie să fie descrise clar, precis, fără ambiguităţi, astfel încât să permită o executare mecanică, automată a acţiunilor algoritmului
   - **Generalitatea** – un algoritm permite, nu rezolvarea unei singure probleme particulare, ci a unei întregi clase de probleme.
   - **Finitudinea** – executarea algoritmului trebuie să cuprindă un număr finit de operaţii, chiar dacă numărul lor este foarte mare. Această proprietate diferenţiază *metoda de calcul* de *algoritm*.
   - **Eficienţa** – dintre algoritmii care rezolvă o anumită problemă, prezintă interes numai algoritmii performanţi pentru care numărul operaţiilor care se execută este cel mai mic.

La momentul actual sunt realizate cercetări serioase asupra algoritmilor care vizează nu numai o îmbunătăţire a performanţei ci şi o reducere a puterii consumate, vitală mai ales la nivelul dispozitivelor de calcul de tip "*handheld*" şi al sistemelor dedicate.

3. **Limbajele de programare** – sunt de două tipuri:
   a) **de nivel înalt** (independente de mașina pe care se procesează – microarhitectura hardware)
      - *C* destinat manipulării structurilor hardware ale calculatoarelor (regiştrii, memorie, porturi)
      - *Fortran* – rezolvarea calculelor ştiințifice
      - *Cobol* – rezolvarea problemelor de procesare a datelor specifice afacerilor (aplicații economice)
      - *Prolog* – folosit în aplicații care necesită proiectarea unui sistem expert.
      - *LISP* – utilizat în rezolvarea problemelor de inteligență artificială.
      - *Pascal* – dezvoltat pentru învățarea studenților să programeze (foarte apropiat de *pseudocod*).
   b) **de nivel scăzut (low –** dependente de mașina pe care se procesează). Există câte un astfel de limbaj specific fiecărei familii de procesoare.
      - *Limbajul de asamblare*.
4. După cum se poate observa şi din figura următoare (fig. 1.5), **Arhitectura Setului de Instrucțiuni (ISA)** – reprezintă interfața dintre software (programele de aplicație / sistem de operare) şi hardware-ul care îl execută. ISA specifică modul de organizare a memoriei (zonă de date statice şi dinamice, de cod, de stivă, zonă rezervată nucleului sistemului de operare), setul de regiştri, setul de instrucțiuni, formatul instrucțiunii, tipurile de date utilizate şi modurile de adresare (mecanismul prin care calculatorul / procesorul localizează operanzii).

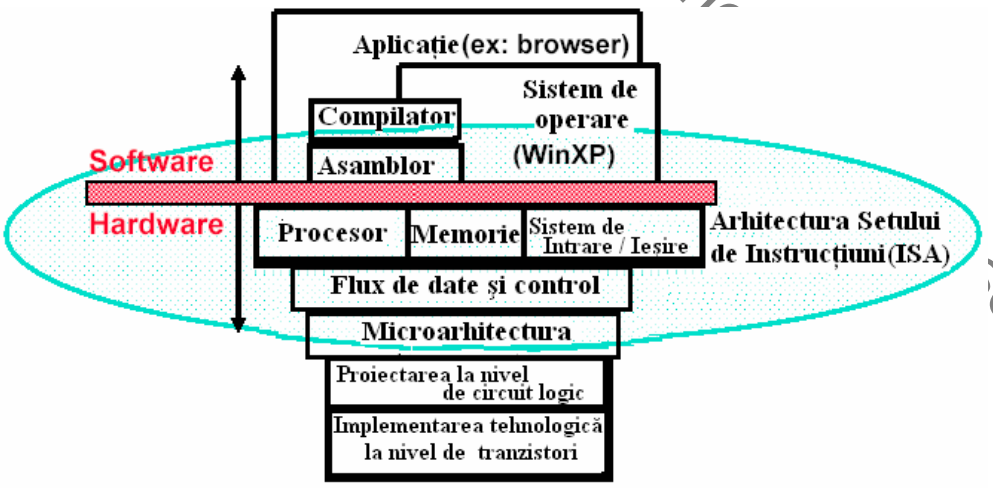


Figura 1.5: Interfața hardware / software

Cele mai cunoscute ISA la nivelul anului 2001 sunt:

- IA-32 ISA (introdusă de Intel Corporation în 1979 şi utilizată în prezent şi de AMD)
- PowerPC (folosite de IBM şi Motorola)
- Alpha (introdusă de Compaq Computer Corporation)
- PA-RISC (utilizată de Hewlett-Packard)
- SPARC (utilizată de SUN Microsystems)

Translatarea unui program de nivel înalt (fie acesta *C*, *Fortran*) **în ISA**-ul **aferent calculatorului care va executa respectivul program** (uzual IA-32) **se realizează prin intermediul compilatorului**. Este denumită *fază* a unui compilator o succesiune de operaţiuni prin care un program de la intrare suferă anumite modificări. Prin *trecere* [Gol97] aparţinând unui compilator se înţelege o citire a programului dintr-un fişier, transformarea lui conform unor faze şi scrierea rezultatului în alt fişier (de ieşire). Se disting două faze majore ale procesului de compilare propriu-zisă:

- **analiza** - în care se identifică părţile constituente fundamentale ale programului şi se construieşte o reprezentare internă a programului original, numită „*cod intermediar*" (***analiza lexicală*** produce un → şir de atomi lexicali -> ***analiza sintactică*** generează -> arborele sintactic → ***analiză semantică*** construieşte o reprezentare a programului sursă în → *cod intermediar*).

- **sinteza** - generează cod maşină eventual optimizat. Se disting două etape:
  - ***optimizare cod intermediar*** (scheduling) pentru o anumită maşină;
  - ***generare de cod maşină*** (generare cod într-o gamă variată de formate: *limbaj maşină absolut*, *limbaj maşină relocabil* sau *limbaj de asamblare* urmat de alocare de resurse). Translatarea din limbaj de asamblare în cod maşină se realizează cu ajutorul unui asamblor.

Pentru fiecare limbaj de programare şi pentru fiecare arhitectură target trebuie să existe un compilator corespondent. Ex: ***gcc*** în sistemul de operare Linux poate transforma un cod sursă C în cod obiect fie pentru procesor Intel, fie MIPS, fie pentru arhitectura virtuală SimpleScalar, în funcţie de bibliotecile şi instrumentele software folosite – asamblor, link-editor, interpretor.

5. **Microarhitectura** – organizarea detaliată a unei implementări de ISA.
    - IA-32 a fost implementată de-a lungul anilor de câteva microprocesoare diferite, fiecare având o microarhitectură unică: de la Intel 8086 în 1979 până la Pentium III în 1999.
    - Alpha ISA este implementată în 4 microprocesoare diferite fiecare cu microarhitectura proprie 21064, 21164, 21264 şi 21364.

Fiecare implementare reprezintă o oportunitate pentru proiectanții de calculatoare de a face diferite compromisuri între costul microprocesoarelor şi performanța acestora. Optimul se obține pe bază de simulare pe benchmark-uri standardizate. ISA descrie funcționalitatea de bază iar microarhitectura care o implementează reprezintă modelul particular obținut în funcție de compromisul cost/performanță (număr stații de rezervare şi unități funcționale de execuție per fiecare tip de instrucțiune, număr de seturi de regiştri generali, valoarea factorului superscalar al microarhitecturii, numărul de faze pipeline de procesare Pentium III – 14, Pentium IV – 20, etc).

De exemplu, **tipul de date** *float* **va afecta numărul de faze pipeline de procesare a unităților de execuție** care operează asupra datelor flotante (de regulă *n* faze, cu n>1), tipului de date *int* corespunzându-i o fază **pipeline.** De asemenea, **setul de regiştrii logici generali sunt implementați fizic la nivelul microarhitecturii**. **Setul de instrucțiuni aferent ISA impune ce tipuri de unități funcționale de execuție / stații de rezervare** specifice să fie implementate la nivel microarhitectural.

6. **Proiectarea la nivel de circuit logic** – implementarea în limbaje de descriere hardware a fiecărui element component al microarhitecturii (regiştri, sumatoare, decodificatoare, multiplexoare, predictoare, cache-uri), urmată de modelare comportamentală şi simulare la nivel de poartă logică pentru a face în final un compromis între cost / complexitate şi performanță. Rezultatul acestei etape îl reprezintă *layout-ul* procesorului care va fi implementat hardware în etapa 7.
7. **Implementarea tehnologică la nivel de circuit**– layout-ul obținut se implementează la nivel de tranzistor prin joncțiuni p-n respectiv n-p (tranzistoare CMOS), polarizate în pastila de siliciu pe mai multe straturi.

Conform celei mai generale clasificări, componentele unui sistem de calcul aparțin uneia dintre următoarele categorii:

- **hardware**-ul, care reprezintă componenta fizică a unui sistem de calcul, în care circuitele electronice asigură prelucrarea automată a

informației precum şi din echipamentele care realizează comunicarea între om şi calculator. Trebuie să asigure cele 4 funcții conform teoriei lui von Neumann (de memorare/ de comandă şi control / de prelucrare / de intrare-ieşire).

- **software**-ul, care reprezintă ansamblul de programe care fac posibilă realizarea funcției sistemului de calcul de prelucrare a datelor şi care constituie suportul logic de funcționare al sistemului de calcul. Se disting două mari componente:
  a) *software de bază* (sistemul de operare) care asigură legătura între componentele fizice şi logice ale calculatorului, având funcții de gestionare a memoriei, a fişierelor, de protecție etc. Inițial se află pe hard-disk iar la pornirea calculatorului se încarcă în memoria principală. Exemple: DOS, Windows, Linux.
  b) *programele de aplicație* (limbaje şi medii de programare, compilatoare, utilitare)– sunt scrise de către programatori la cererea utilizatorilor şi codifică într-un limbaj de programare algoritmul de rezolvare al problemei respective.
- **firmware**-ul, care este componenta de programe încărcate în memoria fixă ROM (Read Only Memory) de către producătorul sistemului de calcul; această componentă se află la limita dintre hardware şi software, reprezentând partea de software integrat în hardware, prin metoda microprogramării.

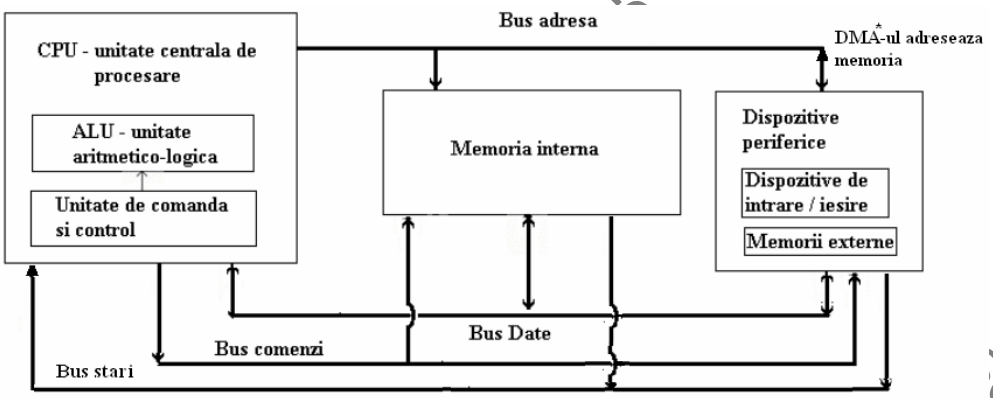


Figura 1.5. Structura unui sistem de calcul

*) **DMA** – dispozitive periferice a căror rată de transfer (octeți /secundă) este atât de ridicată încât, din motive de *timing*, face imposibil modul de lucru prin întreruperi. Ele accesează direct memoria fără intervenția procesorului. Exemple de dispozitive DMA sunt discurile magnetice şi interfețele video.

Unitatea centrală de calcul / procesare (CPU - central processing unit)

- Realizează aducerea din memorie, decodificarea şi execuția instrucțiunilor.
- Dispune de o memorie proprie, foarte mică şi foarte rapidă (Regiştri – 32, 64 de uz general, PC, SP, RA)
- Execută:
  - Operații aritmetice (add/sub/mul/div)
  - Operații logice (and/or/xor/not)
  - Teste de comparație / salturi condiționate
  - Repetat anumite secvențe de instrucțiuni
- Comandă citirea / scrierea din / în memorie, porturi I/O
- Comandă trecerea în stare nedeterminată (*tristate*) a busului de date între procesor şi memorie în cazul DMA.
- Este caracterizat de viteză mare de procesare
- Ceasul procesorului – circuit electronic ce conține un *cristal de cuarț*. Acesta generează impulsuri la intervale regulate stabilindu-se astfel tactul de lucru al procesorului. f=100 MHz $\Rightarrow$ într-o secundă (1s) sunt generate 100 de milioane de impulsuri de tact.

Memoria internă – realizează stocarea programelor şi datelor necesare acestora.

- RAM (memorie cu acces aleator) - are caracter dinamic volatil
- ROM (memorie cu caracter permanent – folosită doar în citire)
- Poate fi statică sau dinamică caz în care un condensator va reprezenta celula inițială de memorare, necesitând reîncărcarea sa periodică (regenerare).
- Unitatea de măsură elementară a memoriei este **bitul** (***Bi****nary Digit***). Definiția bitului: *probabilitatea unui element cu două stări să ia una dintre ele*. O succesiunea de biți din memoria internă poate corespunde unei instrucțiuni, unei date reprezentând valori numerice, text, diverse coduri (imagini, sunet, text).
- Este organizată în diviziuni (locații sau cuvinte) de o mărime egală cu cea a numărului de biți ce poate fi procesată simultan de procesor. O **locație de memorie** este caracterizată de **adresă** şi de **conținut**. Un cuvânt de adresă pe *m* biți poate indexa un spațiu de 2^m locații de memorie.

Bus (magistrală) – interconectează CPU cu memoria şi dispozitivele periferice. Este formată dintr-o mulțime de fire paralele prin care sunt transmise adrese, date, semnale de comandă şi control (Read / Write,

INTerupt Acknowledge), stări ale memoriei sau ale perifericelor (Cereri de întrerupere – semnalul INT, cereri de transfer DMA, Ready – dacă este activ semnifică faptul că Memoria / Dispozitivele de Intrare-iEeşire sunt pregătite pentru transferul de date cu CPU). Magistrala reprezintă practic un set de reguli şi mijloace de a realiza transferul într-un sistem de calcul. Constituie o cale de a transporta informaţii între două dispozitive / echipamente numite sursă şi destinaţie. Din punct de vedere al dialogului pe magistrală (al coordonării transferului de informaţii) modulele implicate se pot afla într-una din următoarele stări:

- *Master*
- *Slave*

Dispozitivele periferice – intermediază comunicaţia calculatorului cu mediul înconjurător. Se disting două clase:

a) **Dispozitive de Intrare / Ieşire** – tastatură, mouse, scanner, microfon (intrare) şi monitor, imprimantă, plotter, difuzor (ieşire).
b) **Memorii externe** – cu caracter nevolatil: hard-disk, floppy disk, CD-ROM, CD-RW. *Sunt mai lente decât memoria internă însă sunt mai ieftine per bit memorat*.

În loc de concluzie:

Cele două "*emisfere*", hardware şi software, în care îşi desfăşoară activitatea cercetătorii din ştiinţa calculatoarelor sunt **doar aparent disjuncte**. Ideea că arhitectura procesoarelor interacţionează "accidental" cu domeniul software este complet greşită, **între hardware şi software existând în realitate o simbioză şi o interdependenţă puternică, încă neexplorate corespunzător**. Procesoarele se proiectează odată cu compilatoarele care le folosesc iar relaţia dintre ele este foarte strânsă:

- **benchmark-urile** (programe de test standardizate) **sunt compilate pentru arhitectura respectivă** iar,
- **compilatorul trebuie să genereze cod care să exploateze caracteristicile arhitecturale**, altfel codul generat va fi ineficient.

Preocupările programatorilor nu trebuie să vizeze doar interfaţa care atrage sau diversele artificii care fac din utilizator un simplu robot ci şi implicaţiile pe care aplicaţia creată o are asupra microarhitecturii. Scopul aplicaţiei trebuie să fie utilizarea cu justeţe atât a resurselor software avute la dispoziţie cât şi a algoritmilor / conceptelor de programare cunoscute (moştenire, polimorfism, apeluri de funcţii prin pointer chiar şi acolo unde nu este cazul). În caz contrar, "*răul*" (a se citi în primul dificultatea procesării rapide datorită ramificaţiilor de program, cod obiect masiv, resurse hardware suplimentare) se răsfrânge asupra performanţelor

arhitecturii. În ce-i privește pe proiectanții de arhitecturi, schemele propuse de aceştia ar putea fi mai eficiente dacă nu ar analiza numai codul obiect al benchmark-urilor avute la dispoziție dezbrăcat de orice semantică ci ar privi "*mai sus*" spre sursa de nivel înalt a acestora.

1.3. EXERCIȚII ŞI PROBLEME

1. Numiți şi explicați noțiunea de algoritm şi trei caracteristici ale acestuia. Pentru fiecare din caracteristici dați un exemplu de procedură care nu o îndeplinește (3 contraexemple).

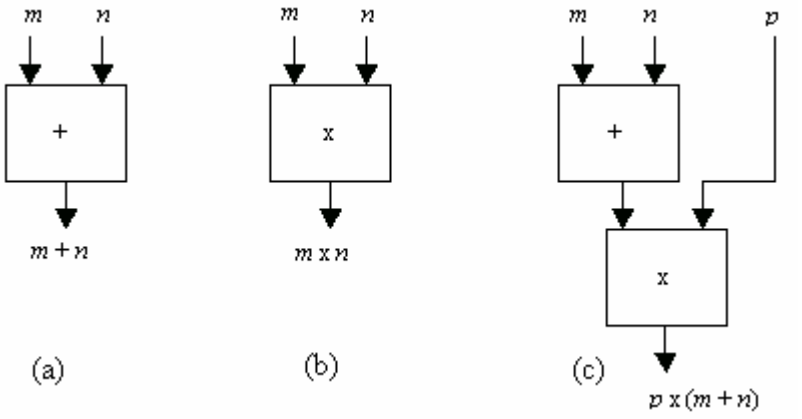
2. Specificați avantajele şi dezavantajele programării într-un limbaj de nivel înalt față de programarea la nivel apropiat de procesor (asamblare).

3. Numiți cel puțin trei noțiuni specificate de ISA (arhitectura setului de instrucțiuni). Descrieți pe scurt diferența dintre ISA şi microarhitectură. Câte ISA sunt implementate în mod normal de o singură microarhitectură. Invers, câte microarhitecturi pot exista pentru o singură ISA ?

4. Enumerați nivelele de transformare şi dați câte un exemplu la fiecare nivel.

5. Care sunt principalele provocări (tendințe) în proiectarea sistemelor de calcul şi ce compromisuri se fac ?

6. Să considerăm următoarea „cutie neagră - *black box*" care preia două numere şi realizează suma lor (vezi figura (a)). Considerăm o altă casetă capabilă să înmulțească două numere (vezi figura (b)). Cele două casete pot fi interconectate pentru a calcula p x (m + n) (vezi figura (c)). Presupunând că dispunem de un număr nelimitat de astfel de casete arătați cum vor fi interconectate pentru a calcula următoarele expresii:
   a) $a \cdot x + b$
   b) Media aritmetică a patru numere w, x, y, şi z.
   c) $a^3 + 3a^2b + 3ab^2 + b^3$ (folosind doar o singură casetă de adunare şi două de înmulțire)



7. Ce a revoluționat tehnologia şi a pus bazele miniaturizării, deschizând calea spre apariția calculatoarelor moderne ? [Log06]

8. Care a fost primul calculător digital de succes, cine sunt constructorii şi unde a fost el utilizat ? Ce operații putea el să execute ? Ce performanțe şi volum avea ? [Log06]

9. Prin ce se caracterizează arhitectura revoluționară propusă de John von Neumann ?

10. Selectați răspunsul corect pentru următorul enunț: „*Ce este memoria RAM din punct de vedere al programatorului ?*” [Zah04]
   a) un dispozitiv de stocare în masă
   b) o zonă de păstrare pentru manipularea / prelucrarea informațiilor curente.
   c) o parte mecanică

11. Selectați continuarea corectă pentru următorul enunț: „*Un algoritm* ” [Zah04]
   a) este o înşiruire de comenzi
   b) este un lanț de instrucțiuni
   c) este o descriere sintetică a unei probleme
   d) rezolvă problema într-un număr finit de paşi
   e) toate variantele de mai sus sunt corecte.

2. REPREZENTAREA INFORMAȚIILOR. CONVERSII DE VALORI ÎNTRE SISTEME DE NUMERAȚIE. ARITMETICĂ BINARĂ. OPERAȚII LOGICE

2.1. REPREZENTAREA INFORMAȚIILOR ÎN SISTEMELE DE CALCUL

În acest capitol se urmăreşte identificarea tipurilor de informaţii reprezentate într-un calculator şi definirea modurilor de codificare a acestora în vederea memorării, prelucrării şi transmiterii. Se vor efectua conversii de date între sistemele de numeraţie binar, zecimal şi hexazecimal. Se vor reprezenta numere pozitive şi negative în semn şi mărime, complement faţă de 1 şi complement faţă de 2; se vor comenta avantajele şi dezavantajele celor 3 variante de reprezentare. Se vor efectua operaţii aritmetico-logice cu valori numerice reprezentate în cod complementar. Se vor reprezenta numere în virgulă flotantă; se vor analiza limitările acestei forme de reprezentare şi se va face o comparaţie cu reprezentarea în virgulă fixă.

Informaţia – Definiţie. Cuantificare. Convenţii de memorare în calculator. Clasificare.

- **Definirea informaţiei**
  - noţiune primară - greu de definit, reprezintă un atribut al materiei
  - ***"mesaj" obiectiv care aduce o clarificare privind starea unui proces care are mai multe stări şi elimină nedeterminarea în legătură cu realizarea unui eveniment***.
- **Cantitatea de informaţie** - *entropia informaţională* (***E***)
  - Claude E. Shannon [Sha48] a introdus noţiunea de **bit** (***bi****nary digit*) ca fiind „**unitatea de măsură a informaţiei**". Shannon afirmă că există o corelaţie între **informaţia produsă prin apariţia unui eveniment în cadrul unui experiment şi logaritmul inversului probabilităţii de apariţie a acestui eveniment**. Plecând de la această definiţie, Shannon a introdus conceptual de entropie

informațională (E), element de maximă importanță în teoria informației şi a ştiinţei comunicațiilor.

- ***E*** reprezintă o măsură a gradului de incertitudine (nedeterminare) dintr-un sistem. Este dependentă de numărul maxim de stări posibile al respectivului sistem. Gradul de organizare a sistemului este direct proporțional cu cantitatea de informație înmagazinată şi invers proporțional cu entropia informațională a sistemului.
- Exemple: aruncarea unei monede, aruncarea unui zar

$$\begin{bmatrix} x_1 & x_2 & \dots & x_n \\ p_1 & p_2 & \dots & p_n \end{bmatrix} \qquad E = -\sum p_i \log_2 p_i$$

Observație: Entropia devine maximă (*dezordine maximă*) pentru o distribuție echiprobabilă. Astfel dacă $p_i = \frac{1}{n}, \forall i = \overline{1,n}, \Rightarrow E = \max = \log_2 n$ – valoare ce reprezintă numărul minim de biți necesar codificării entropiei. Analog, entropia devine minimă (*ordine maximă*) când $p_k = 1$ si $p_i = 0 \ \forall i \neq k, i = \overline{1,n}, \Rightarrow E = \min = 0$, considerând prin convenție $\log_2 0 = 0$.

De exemplu, pentru 2 stări echiprobabile entropia este:

$E = - 2*0.5*\log_2 0.5 = 1$ unitatea de informație (1 bit). O înşiruire de 8 biți formează un octet (1 byte). Multiplii octeților sunt **kilo/mega/giga/tera/peta** octeții:

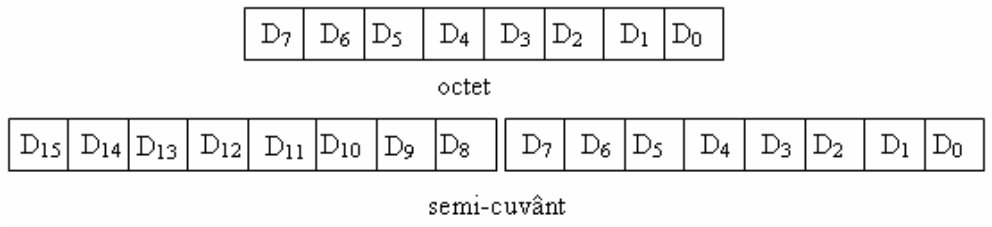
- **1ko** (kilo octet) = 2^{10}(1.024)o ≈ 10^3o
- **1Mo** (mega octet) = 2^{20}(1.048.576)o ≈ 10^6o
- **1Go** (giga octet) = 2^{30}(1.073.741.824)o ≈ 10^9o
- **1To** (tera octet) = 2^{40}(1099511627776)o ≈ 10^{12}o
- **1Po** (peta octet) = 2^{50}(1099511627776)o ≈ 10^{15}o

■ Convenții de memorare a informației în calculator

Memoria internă a unui calculator este alcătuită dintr-o succesiune de locații; fiecare locație are un număr fix de biți, număr stabilit de către proiectantul de arhitectură în momentul în care se decide modul de organizare a memoriei. Astfel, în mod uzual numărul de biți ai unei locații este un multiplu de 8. Locația este unitatea elementară de adresare a unei memorii; la majoritatea procesoarelor (cu excepția microcontrolerelor) informația din memorie nu se poate adresa la nivel de bit. Indexarea locațiilor de memorie se face, pe bază de adresă. La procesoarele Intel, MIPS unitatea minimă de adresare (locația de memorie) are 8 biți. În acelaşi

timp însă memoria se poate adresa la nivel de semi-cuvânt (*halfword* – 16 biți), cuvânt (*word* – 32 biți) sau dublu-cuvânt (*double* – 64 biți).

Ca în cazul oricărui sistem de numerație pozițional (8,10,16) şi în cazul sistemului binar bitul **cel mai puțin semnificativ** al unui octet (bitul D_0) se află în poziția cea mai din dreapta a datei, iar **bitul cel mai semnificativ** în poziția stângă. Importanța acestei convenții se remarcă în cazul operațiilor logice de deplasare şi rotire spre dreapta sau spre stânga.



Dacă o dată este reprezentată pe un număr mai mare de octeți atunci se pune problema modului de amplasare a octeților la adrese consecutive de memorie (așa numită *ordine a octeților*). Procesoarele pot numerota octeții din interiorul unui cuvânt de 32 de biți astfel încât octetul cu numărul cel mai mic este fie cel mai din stânga fie cel mai din dreapta. Există două convenții de reprezentare:

- *big endian* – partea mai semnificativă a numărului se pune la adrese mai mici (se pune în față).

Byte #			
0	1	2	3

- *little endian* – partea mai puțin semnificativă a datei se amplasează la adrese mai mici (se pune în față).

Byte #			
3	2	1	0

De exemplu în convenția *little endian* o valoare hexazecimală de forma 11223344h se memorează la adresa fizică 400h în felul următor:

De exemplu, procesoarele Alpha, Intel şi MIPS folosesc convenţia "*little-endian*", în timp ce în cazul procesoarelor MacIntosh, HP Bobcat sau SPARC ordinea folosită este "*big-endian*".

■ **Clasificare**

Se reaminteşte că una din funcţiile de bază ale calculatorului este de prelucrare a informaţiilor. Pentru a putea fi prelucrate, informaţiile trebuie reprezentate (codificate) într-un anumit format, pe baza unor reguli bine definite şi lipsite de ambiguităţi. Modul de reprezentare a informaţiilor depinde de tipul informaţiilor şi de obiectivele prioritare urmărite.

Informaţiile prelucrabile pe un calculator sunt de mai multe tipuri:

- program (executabil) – secvenţă de coduri de instrucţiuni
- date:
  - numerice:
    - întregi
      - fără semn – numere naturale
      - cu semn – numere întregi
    - reale – numere cu parte întreagă şi parte fracţionară (practic sunt numere din mulţimea Q a numerelor fracţionare)
  - alfanumerice – text
  - logice – adevărat/fals
  - multimedia
    - audio
    - video
      - imagini statice
      - secvenţă de cadre (film)
  - semnale – mărimi fizice de proces, detectate prin senzori sau transmise prin elemente de acţionare

Interschimbarea noţiunilor „**dată"** respectiv „**informaţie"** este larg răspândită în lumea largă, nefăcându-se distincţie de cele mai multe ori între ele, fiind tratate ca şi sinonime. În realitate însă [Bel04], există diferenţe între cele două noţiuni: **informaţia are un conţinut semantic** (obţinut în urma unui proces de interpretare), pe când **data este forma fizică de reprezentare a unei informaţii**. Potrivit cercetătorului Russell Ackoff [Ack89] conţinutul minţii umane poate fi clasificat în 5 categorii (fiecare categorie bazându-se pe cele de dinaintea sa): *date*, *informaţii*, *cunoaştere*, *înţelegere* şi *înţelepciune*. Se pune întrebarea ce decurge din aceste observaţii: *Calculatorul prelucrează date sau informaţii*? Dacă se consideră în mod simplist că un program specifică o secvenţă de transformări prin care

trec datele inițiale în scopul generării unui rezultat, atunci se poate spune că un calculator prelucrează date. Prin interpretarea datelor un utilizator le transformă însă în informație. De exemplu data 49 poate fi o valoare de temperatură, o vârstă, o poziție într-un şir, o dimensiune sau codul ASCII al cifrei 1 (49=0x31) văzut ca şi caracter ('1'). Există însă şi cazuri, mai ales în domeniul inteligenței artificiale [Seb, Gor05], în care calculatorul identifică şi percepe anumite concepte sau relații, care nu mai sunt simple date.

Pentru fiecare tip de informație [Seb] există una sau mai multe forme de reprezentare (codificare). Alegerea unei anumite forme de codificare se face în funcție de anumite obiective urmărite, cum ar fi:

- reprezentarea coerentă, a informațiilor în vederea stocării, transmiterii şi a prelucrării acestora
- utilizarea eficientă a spațiului alocat (spațiu minim)
- detecția şi corecția erorilor
- simplificarea operațiilor de prelucrare, stocare şi transmitere
- securizarea datelor

2.1.1. REPREZENTAREA INFORMAȚIILOR NUMERICE FĂRĂ SEMN. SISTEME DE NUMERAȚIE

Pentru **reprezentarea valorilor numerice** se pot folosi diferite **sisteme de numerație** [Gor05, Seb, Mâr96]. Un astfel de sistem **este definit printr-un *set finit de simboluri şi un set de reguli* pentru reprezentarea valorilor prin combinații de simboluri (**numere şi litere**)**. Se cunosc două tipuri de sisteme de numerație: *ponderate* (Ex: binar, zecimal, hexazecimal) şi neponderate (Ex: codul Gray). Un *sistem ponderat de numerație* atribuie ponderi pentru fiecare poziție dintr-o reprezentare. De regulă, ponderile sunt puteri crescătoare ale unei valori; această valoare considerându-se baza sistemului ponderat de numerație. Un sistem de numerație ponderat se defineşte în felul următor:

- *simboluri*: 0, 1, 2, 3, ... (b-1) (1), unde *b* reprezintă baza sistemului de numerație
- *reprezentare*: $N_b \rightarrow x_n...x_2x_1x_0$, $x_n \neq 0$ (2)
- *calculul valorii*: $N_b = x_n*b^n+...x_2*b^2+x_1*b^1+x_0$ (3)

Codurile neponderate asociază fiecărei cifre zecimale o tetradă binară, cifrele binare neavând însă semnificația unor ponderi. Codul Gray are

caracteristic faptul că fiecare cifră zecimală, reprezentată pe 4 cifre binare, diferă de următoarea cifră zecimală, prin modificarea unei singure cifre binare din tetradă şi a fost introdus pentru a minimiza o serie de erori în proiectarea automatelor.

Gray	Zecimal
0000	0
0001	1
0011	2
0010	3
0110	4
0111	5
0101	6
0100	7
1100	8
1101	9

Tabelul 2.1. Echivalența cod Gray – zecimal

Sistemul de numerație binar este practic cel mai potrivit pentru reprezentarea valorilor numerice într-un calculator. Există două explicații pentru această alegere:

- *Simplitatea implementării* folosind circuite care au 2 stări stabile corespunzătoare celor două simboluri ale sistemului binar 0 şi 1.
- *Reguli de efectuare a operațiilor aritmetice mult mai puține* decât în oricare alt sistem de numerație, ceea ce simplifică structura unității aritmetice.

În sistemul binar pentru reprezentarea întregilor fără semn (naturale) *b* din (1) este egal cu 2. Pentru numerele întregi fără semn pe **n** biți pot reține $\mathbf{2^n}$ numere în intervalul $\mathbf{0 \div 2^n - 1}$. Dacă **A** este un întreg fără semn rezultă că numărul de biți pe care poate fi reprezentat este: $\mathbf{n_b = [\log_2 A] + 1}$. Demonstrația se bazează pe faptul că orice număr natural (fie A în acest caz) poate fi încadrat între două puteri consecutive ale lui 2 ($2^{n_b-1} \leq \mathbf{A} < 2^{n_b}$).

Dezavantajul sistemului binar de numerație constă în numărul mare de cifre necesare pentru reprezentarea unei anumite valori (mai mare decât în orice alt sisteme de numerație).

Pentru reprezentarea numerelor cu parte fracționară s-a introdus încă un simbol suplimentar, şi anume punctul zecimal. Toate cifrele care urmează după punctul zecimal sunt ponderate cu puteri negative ale bazei, după cum urmează:

- reprezentare: $N_b \rightarrow x_n...x_2x_1x_0 \bullet x_{-1}x_{-2}...x_{-m}$
- calculul valorii:

$$N_b = x_n*b^n+...x_2*b^2+x_1*b^1+x_0*b^0+ x_{-1}*b^{-1}+x_{-2}*b^{-2}+.... x_{-m}*b^{-m}$$

Conversii între sisteme de numerație

a) Din zecimal în binar

Conversia unui număr dintr-o bază de numerație în alta se face separat pentru partea întreagă şi pentru partea fracționară. *Partea întreagă se divide succesiv cu noua bază şi se reţin resturile parțiale în ordinea inversă a generării lor. Partea fracționară se înmulțeşte succesiv cu baza şi se reține de fiecare dată partea întreagă a rezultatului*. Corectitudinea acestor tehnici de conversie se demonstrează matematic.

- Partea întreagă

$N= (...(x_m+0)b+x_{m-1})b+...+x_1)b+x_0$
$N=N_1 \cdot b+x_0$
$N_1=N_2 \cdot b+x_1$
...
$N_m = 0*b+x_m$

- Partea fracționară

$N = N_{\hat{i}ntr.}+N_{zec.}$
$N_{zec} = x_{-1} \cdot b^{-1}+x_{-2} \cdot b^{-2}+ ...$
$x_{-1} = [N_{zec} \cdot b]$
$N_{zec} \cdot b = x_{-1} +N_{2zec}$
$x_{-2}= [N_{2zec} \cdot b]$
.....

De exemplu pentru conversia numărului 23.45_{10} din zecimal în binar se fac următoarele operații:

23		$23_{10} = 10111_2$	0,45		$0,45_{10}=0,01110..._2$
11	1		0,9	0	
5	1		1,8	1	
2	1		1,6	1	
1	0		1,2	1	
0	1		0,4... ..	0	

Se observă că în cazul conversiei părţii fracţionare de cele mai multe ori nu se obţine un rezultat exact. Procesul de conversie se încheie atunci când se obţine o precizie rezonabilă de exprimare a valorii fracţionare. De exemplu în multe aplicaţii inginereşti o precizie de 2 cifre zecimale după virgulă se consideră o precizie acceptabilă. Întrebarea este câte cifre binare sunt necesare pentru o precizie similară? La exprimarea în baza zece, prin două cifre după punctul zecimal se obţine o eroare maximă de 1/100. În baza 2 pentru o precizie similară trebuie să se utilizeze 7 cifre binare pentru a obţine o eroare maximă de 1/128, adică $1/2^7$. În mod similar pentru o precizie mai mare de 1/1000 (3 cifre zecimale după punct) sunt necesare 10 cifre binare ($1/1024 = 1/2^{10}$).

Poziţia punctului zecimal nu se reprezintă în calculator. Prin convenţie se consideră într-o poziţie predefinită. Pentru reprezentarea numerelor strict întregi poziţia punctului zecimal se consideră în dreapta reprezentării (după cifra cea mai puţin semnificativă). Din această cauză această codificare poartă numele de "***reprezentare în virgulă fixă***". Pentru operaţiile de adunare şi scădere poziţia punctului zecimal nu influenţează rezultatul generat. Situaţia este diferită pentru operaţiile de înmulţire şi împărţire, unde poziţia punctului în rezultat se schimbă.

b) Din binar în hexazecimal

Pentru a face trecerea de la sistemul zecimal la cel binar şi invers adesea se utilizează un sistem de numeraţie intermediar, care se apropie de ambele sisteme. De exemplu sistemul hexazecimal este utilizat în acest scop deoarece, pe de-o parte permite exprimarea unor valori printr-un număr redus de cifre, iar pe de altă parte se poate converti relativ simplu în sistemul binar. În cazul în care se vizualizează conţinutul unei zone de memorie sau a unor regiştrii informaţia este prezentată în format hexazecimal, chiar dacă în calculator informaţia există în binar. Sistemul octal şi ulterior hexazecimal au fost introduse şi datorită existenţei afişoarelor cu 7 segmente.

Convertirea unui număr din sistemul binar în sistemul hexazecimal şi invers (vezi tabelul 2.2) se face prin gruparea biţilor câte 4 (cele 16 cifre, valori hexazecimale - sunt reprezentate pe 4 biţi), începând de la punctul zecimal spre dreapta şi spre stânga şi echivalarea câte unei cifre hexazecimale cu o combinaţie de 4 biţi.

Binar	Hexazecimal	Zecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Tabelul 2.2. Echivalența binar – hexazecimal –zecimal.

Exemple de conversie binar – hexazecimal şi invers:

$1011.0011\bullet1100.11_2$ => $B3\bullet CC_{16}$ $1FD_{16}$ => $1.1111.1101_2$

$101\bullet1111.011_2$ => $5\bullet F6_{16}$ $33\bullet3_{16}$ => $11.0011\bullet0011_2$

2.1.2. REPREZENTAREA NUMERELOR CU SEMN

Pentru reprezentarea numerelor pozitive şi negative se alocă un bit suplimentar, care indică semnul. Există 3 forme de reprezentare a acestor numere:

- prin semn şi magnitudine (mărime) sau *cod direct*
- în complement față de unu sau *cod invers*
- în complement față de 2 sau *cod complementar*

Indiferent de reprezentare, bitul de semn este egal cu 0 dacă numărul este pozitiv şi 1 dacă numărul este negativ. Pentru evitarea ambiguităților, la acest tip de reprezentare este important să se stabilească de la început numărul de biți pe care se face reprezentarea.

Reprezentarea prin semn şi magnitudine (SM) – *cod direct*:

Se reprezintă valoarea absolută a numărului pe n-1 poziţii binare (n – numărul total de biţi ai reprezentării) şi se adaugă un bit de semn (**0** - pt. numere **pozitiv**e şi **1** pt. numere **negativ**e) pe poziţia cea mai semnificativă a reprezentării. De exemplu:

-21_{10} -> **1**001.0101$_2$ 17_{10} -> **0**001.0001$_2$ 0_{10} -> 0000.0000$_2$ (reprezentări pe 8 biţi)

129_{10} -> valoarea absolută nu se poate reprezenta pe 7 biţi (ca în cele trei exemple anterioare) => reprezentarea se poate face numai pe un număr mai mare de biţi (minim 9)

129_{10} -> **0**000.0000.1000.0001$_2$ -129_{10} -> **1**000.0000.1000.0001$_2$ (reprezentări pe 16 biţi)

Deşi această **formă de reprezentare** este **simplă,** ea **nu este avantajoasă pentru implementarea operaţiilor aritmetice întrucât unitatea aritmetico-logică ar trebui să ţină cont de semnul operanzilor**.

Reprezentarea în complement faţă de 1 (C1) – *cod invers*:

Numerele pozitive se reprezintă ca şi în cazul anterior, adică n-1 biţi pentru valoare şi 1 bit, egal cu 0 pentru semn. În cazul numerelor negative se complementează fiecare poziţie binară a reprezentării valorii absolute a numărului, inclusiv bitul de semn. De exemplu:

$|-24_{10}|$ -> **0**001.1000 19_{10} -> 0001.0011 $|-127_{10}|$ -> 0111.1111
-24_{10} -> **1**110.0111 0001.0011 -127_{10} -> 1000.0000

Se observă că -127 + 127 = 1111.1111. Dar $-x+x=0$ = 0000.0000 $\forall x \in R \Rightarrow$ **Dezavantaj:** ***Există două reprezentări ale numărului 0.***

Reprezentarea în complement faţă de 2 (C2) - *cod complementar*:

Numerele pozitive se reprezintă ca şi în cazurile anterioare. Reprezentarea numerelor negative se obţine prin adăugarea unei unităţi la reprezentarea în complement faţă de 1. De exemplu:

$|-21_{10}|$ -> 0001.0101 15_{10} -> 0000.1111 $|-112_{10}|$ -> 0111.0000

-21_{10} -> **1**110.1010+ C1 **0**000.1111 C1 -112_{10} -> **1**000.1111+ C1
1 1
-21_{10} -> **1**110.1011 C2 **0**000.1111 C2 -112_{10} -> **1**001.0000 C2

Reprezentarea în complement față de 2 este avantajoasă deoarece simplifică modul de calcul al operațiilor aritmetice. La operațiile de adunare şi scădere unitatea aritmetică nu trebuie să țină cont de semnul operanzilor, rezultatul generat fiind corect indiferent de semn. Mai mult rezultatul este corect şi în cazul în care se consideră că reprezentările sunt numere strict pozitive. Pe baza acestei observații se poate utiliza aceeaşi unitate de adunare/scădere atât pentru numere pozitive cât şi pentru numere cu semn. De exemplu:

Numere cu semn		Numere fără semn
-64+	1100.0000+	192+
15	0000.1111	15
-49	1100.1111 ->207	207
	0011.0001 -> 49	

În ziua de azi toate calculatoarele utilizează reprezentarea în complement față de 2. Celelalte două forme s-au folosit la primele calculatoare.

Observație!!! Reprezentarea în cod complementar a unui număr pozitiv nu înseamnă complementarea numărului, ci reprezentarea sa în formă pozitivă, care este identică în cele trei forme de reprezentare (SM, C1 şi C2).

La reprezentarea în complement față de 2, dacă se trece de la o reprezentare pe 8 biți la una pe 16 biți sau mai mare atunci este necesară **extinderea semnului**, în aşa fel încât în noua reprezentare valoarea şi semnul numărului să se păstreze. Regula constă în copierea valorii bitului de semn în fiecare poziție binară a porțiunii extinse. De exemplu:

Valoare	Reprezentare C2 pe 8 biți	Reprezentare C2 pe 16 biți
-6	**1**111.1010	**1111.1111**.1111.1010
0	**0**000.0000	**0000.0000**.0000.0000
6	**0**000.0110	**0000.0000**.0000.0110

Tabelul 2.3. Extinderea semnului la reprezentarea pe mai mulți biți decât sunt necesari

Conversia din binar în zecimal a numerelor întregi cu semn

Considerând o valoare reprezentată în cod complementar pe 8 biți ($a_7a_6…a_1a_0$) rezultă că valoarea corespondentă în zecimal aparține intervalului $[-2^7, 2^7-1]$. Conversia se realizează în următorii paşi:

1. Se examinează cel mai semnificativ bit.
   - Dacă a_7=0 atunci numărul reprezentat este pozitiv şi se poate trece la determinarea magnitudinii sale (vezi pasul 2).
   - Dacă a_7=1 atunci numărul reprezentat este negativ. În acest caz se determină reprezentarea în cod complementar a numărului pozitiv care are aceeaşi magnitudine şi se calculează mărimea acestuia.
2. Magnitudinea (valoarea absolută) se determină cu formula:
   $a_6 \cdot 2^6 + a_5 \cdot 2^5 + \ldots + a_0 \cdot 2^0 \cdot$(practic se adaugă puterile lui 2 acolo unde coeficienţii $a_i \cdot$sunt 1).
3. În final dacă numărul iniţial este negativ atunci valorii absolute determinate la punctul 2 îi este prefixat simbolul '–'.

Exemplu:

Se dă exprimat în cod complementar numărul: 11000111. Să se determine valoarea lui în zecimal.

Întrucât cel mai semnificativ bit este 1 rezultă că numărul este negativ. Se determină reprezentarea în cod complementar a numărului pozitiv care are aceeaşi magnitudine şi anume: 00111000+1=00111001. Magnitudinea acestuia este: $2^5 + 2^4 + 2^3 + 2^0 = 57$. Numărul iniţial (11000111) fiind negativ rezultă că valoarea întreagă corespondentă în zecimal acestuia este –57.

2.1.3. REPREZENTAREA NUMERELOR ÎN VIRGULĂ FLOTANTĂ. FORMATUL IEEE 754

Această formă de codificare se utilizează pentru reprezentarea unor valori foarte mari, foarte mici sau dacă numerele au o parte fracţionară. Pentru reprezentarea în virgulă flotantă numărul este adus la o formă normalizată (standard) în care există o parte strict subunitară (mantisa) care se înmulţeşte cu o putere a lui 2 (exponentul). La mantisă prima cifră de după virgulă este strict diferită de 0.

$$N = \pm 1,f \times 2^{\pm e}$$

unde,

N – este numărul reprezentat în virgulă mobilă

f – reprezintă partea fracţionară a lui N

$\pm 1,f$ – se numeşte mantisă şi trebuie să respecte relaţia de normalizare:

$$1 \le |\pm 1,f| < 2$$

2 – baza sistemului de numerație

±e – reprezintă exponentul bazei sistemului de numerație

IEEE *(Institute of Electrical and Electronics Engineer*s) a dezvoltat un standard pentru reprezentarea numerelor în virgulă mobilă şi operațiile aritmetice în această reprezentare. Scopul era facilitarea portabilității programelor între diferite calculatoare. Standardul IEEE 754 a fost publicat în 1985. Cele mai multe coprocesoare aritmetice, printre care şi cele *Intel* pentru familia de microprocesoare 80x86, se conformează acestui standard. În reprezentarea standard IEEE se folosesc 3 câmpuri (vezi tabelul 2.4):

- un bit de semn (S) – indică semnul numărului
- un câmp pentru exponent (*caracteristică*) – indică magnitudinea numărului
- un câmp pentru mantisă *(fracție)* – indică un set de cifre semnificative ale numărului

31	**30…….……23**	**22…….……0**
S	Caracteristică	Fracție

Tabelul 2.4. Reprezentarea unui număr în virgulă mobilă simplă precizie

Standardul defineşte trei formate:

- Formatul *scurt* (simplă precizie) – 32 biți – 1 semn + 8 caracteristică + 23 mantisă
- Formatul *lung* (dublă precizie) – 64 biți – 1 semn + 11 caracteristică + 52 mantisă
- Formatul *temporar* (precizie extinsă) – 80 biți – 1 semn + 24 caracteristică + 56 mantisă

Caracteristica reprezintă exponentul deplasat. Pentru formatul scurt, deplasamentul este 127 (7Fh), iar pentru formatul lung deplasamentul este 1023 (3FFh). Valorile minime (0) şi cele maxime (255, respectiv 2047) ale caracteristicii nu sunt utilizate pentru numerele normalizate, ele având utilizări speciale.

$$NS = (-1)^S \times Mantisă \times 2^{\text{Caracteristică-127}}$$

$$ND = (-1)^S \times Mantisă \times 2^{\text{Caracteristică-1023}}$$

Gama numerelor care pot fi reprezentate în precizie simplă este cuprinsă între aproximativ $2{,}2\times10^{-38}$ şi $3{,}4\times10^{38}$, iar cea a numerelor reprezentate în precizie dublă este cuprins între $2{,}2\times10^{-308}$ şi $1{,}7\times10^{308}$.

Mulți echivalează în mod greşit reprezentarea în virgulă flotantă cu mulțimea numerelor reale din matematică. Există câteva deosebiri esențiale:

- în virgulă flotantă se reprezintă valori discrete şi nu domeniu continuu de valori (submulțimi de numere raționale a mulțimii numerelor reale)
- nu pot fi reprezentate numere foarte mari sau foarte mici; de exemplu sunt probleme în ceea ce priveşte reprezentarea valorii zero şi a valorilor +/- ∞
- rezoluția absolută de reprezentare variază cu valoarea reprezentată
- compararea a două valori flotante poate genera surprize datorită preciziei de reprezentare şi a rotunjirilor

Operații efectuate în virgulă mobilă

- *Adunarea şi scăderea* a două numere în virgulă mobilă se efectuează astfel:
  - ✓ Se compară cei doi exponenți pentru a-l determina pe cel mai mare.
  - ✓ Se aliniază mantisa numărului cu exponent mai mic, prin deplasarea virgulei corespunzătoare exponentului mai mare.
  - ✓ Se adună / scad mantisele aliniate, atribuind exponentul comun.
  - ✓ Eventual se normalizează mantisa concomitent cu modificarea exponentului.
- *Înmulțirea şi împărțirea* presupun:
  - ✓ Adunarea (scăderea) exponenților.
  - ✓ Înmulțirea (împărțirea) mantiselor.
  - ✓ Eventuala normalizare a mantiselor.

Exemple [Mâr96]:
Să se reprezinte numerele în virgulă mobilă simplă precizie:

a) $25_{10} = (11001)_2 = 1{,}1001 \times 2^4$

unde:

mantisa = 1,1001
fracția = 1001
exponentul = 4
Semnul = 0
Caracteristica = exponentul + 127 = 131

31	**30…….23**	**22……..……0**
0	10000011	100100.....0

b) $\frac{31}{32} = \frac{2^4+2^3+2^2+2^1+2^0}{2^5} = 2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5} = 0{,}11111_2 = 1{,}1111_2 \times 2^{-1}$

unde:

mantisa = 1,1111
fracția = 1111
exponentul = -1
Semnul = 0
Caracteristica = exponentul + 127 = 126

31	30…….23	22……..……0
0	01111110	11110.....0

c) $-\frac{1}{8} = -\frac{2^0}{2^3} = -2^{-3} = -0{,}001_2 = -1{,}0_2 \times 2^{-3}$

unde:

mantisa = –1,0
fracția = 0
exponentul = -3
Semnul = 1
Caracteristica = exponentul + 127 = 124

31	30…….23	22……..……0
1	01111100	00000.....0

Dacă se urmărește acum adunarea a două din cele trei numere exprimate mai sus: 25 - 1/8 rezultă se obține:
25 - 1/8 = $1{,}1001 \times 2^4 - 1{,}0 \times 2^{-3} = 1{,}1001 \times 2^4 - 0{,}0000001 \times 2^4 = 1{,}1000111 \times 2^4 = 11000{,}111_2$ = **24,875**

Reprezentarea în virgulă mobilă a rezultatului este:

31	30…….23	22……..……0
0	10000011	10001110.....0

2.1.4. REPREZENTAREA INFORMAȚIILOR ALFANUMERICE – STANDARDUL ASCII

Informațiile alfanumerice sunt cele care se prezintă sub formă de text şi eventual conțin o grafică simplă bazată pe forme grafice predefinite de dimensiunea unui caracter. Cel mai utilizat sistem pentru reprezentarea informațiilor alfanumerice este standardul **ASCII** (American Standard Coding for Information Interchange). Acest standard utilizează 7 biți pentru reprezentarea codurilor alfanumerice (8 biți în varianta extinsă). Sunt codificate următoarele tipuri de date:

- litere mari şi mici
- cifre zecimale
- semne de punctuație
- coduri de editare şi formatare a textului
- coduri de control al transferului de date

Tabelul 2.5 conține câteva exemple de coduri ASCII:

Litere		**Cifre**	
A	41h	0	30h
B	42h	1	31h
C	43h	2	32h
....		...	
a	61h	9	39h
b	62h	**Coduri de formatare**	
c	63h	spațiu	20h
....		CR- retur	0Dh
Semne de punctuație		LF – linie nouă	0Ah
.	2Eh		
,	2Ch	**Coduri de control al transferului**	
Operații aritmetice		XON- pornire transmisie	11h
+	2Bh	XOFF- oprire transmisie	13h
-	2Dh		

Tabelul 2.5. Reprezentarea unui număr restrâns de caractere din codul ASCII

Standardul "*unicode*" extinde codificarea ASCII prin adăugarea unui octet suplimentar. Astfel anumite litere speciale (litere grecești, litere arabe, litere cu semne speciale, etc.) sau semne grafice sunt reprezentate pe 16 biți.

2.1.5. REPREZENTAREA INFORMAȚIILOR MULTIMEDIA

Se spune că o imagine valorează cât 1000 de cuvinte. Adevărul este că pentru reprezentarea unor informații multimedia spațiul necesar pentru memorare este semnificativ mai mare decât pentru alte tipuri de informație. De exemplu pentru memorarea unei imagini spațiul de memorie variază (funcție de rezoluție) între câteva sute de kocteți şi câțiva megaokteți (vezi tabelul 2.6).

Tip de informație	Detalii	Cantitatea de informație
logică	1 semnal bipozițional	1 bit
numerică	1 întreg	16-32 biți
text	1 pagină	~1 KOctet
audio	10 sec., Fmax=20KHz	400 KOcteți
video	10 sec. Fcadre=50Hz÷100Hz Rez.=1000*1000	1,5 GOcteți

Tabelul 2.6**. Cantitatea de informație (spațiu necesar pentru stocare)**

Informațiile multimedia (imagini şi/sau sunete) **sunt reprezentate prin eşantioane ale semnalului de intrare.** Frecvența de eșantionare trebuie să fie în strictă corelație cu frecvența maximă a semnalului de intrare. Conform teoremei lui Shanonn ***frecvența de eșantionare trebuie să fie de cel puțin două ori mai mare decât maximul frecvenței semnalului de intrare*** [Nic04]. În caz contrar reprezentarea nu este fidelă cu realitatea; apar frecvențe inexistente în semnalul inițial (fenomenul de "aliasing"). În general, se utilizează o frecvență a semnalului de eşantionare cam de 10 ori frecvența semnalului de intrare, minim de 5 ori din motive de cost (pentru costuri reduse) şi se poate ajunge chiar şi la 100 ori.

Semnalul audio este eşantionat în timp cu o frecvență cuprinsă între 8kHz şi 50KHz [Seb, Gor05], funcție de calitatea reprezentării. Frecvența de 8KHz este acceptabilă pentru convorbiri telefonice (unde lățimea de bandă este limitată la 4 KHz), iar frecvențe de 40-50KHz sunt necesare pentru semnale audio de înaltă fidelitate (urechea umană detectează semnale de maxim 16-20kHz). Pentru un eşantion se alocă de obicei 8 biți asigurându-se astfel o plajă de valori pentru variația de amplitudine de 256 de valori,

suficientă pentru majoritatea aplicațiilor uzuale. În aceste condiții pentru memorarea unui semnal sonor de 10 secunde este nevoie de 80-500Kocteți. Pentru reducerea spațiului de memorare se folosesc diferite tehnici de compresie, prin care acest spațiu se poate reduce până la aproximativ o zecime din valoarea inițială. Majoritatea schemelor de compresie sunt cu pierdere de informație, adică nu garantează refacerea fidelă a semnalului sonor în urma decompactării. Dar pentru acest tip de informație acest fapt nu este un impediment, mai mult chiar, algoritmii de compactare au şi un efect de filtrare a semnalului audio.

În cazul informațiilor video se utilizează tot un proces de eşantionare în timp a semnalului video de intrare. În acest mod **informația video este împărțită pe puncte (pixeli), linii şi cadre**. **O imagine are o** anumită **rezoluție dimensională** şi **cromatică**. *Rezoluția dimensională se defineşte pe două direcții*:

- *orizontală* – număr de pixeli pe o linie şi
- *verticală* – număr de linii pe un cadru

Rezoluția cromatică este determinată de numărul de biți alocați pentru reprezentarea unui pixel. La **imaginile alb/negru numărul de biți alocați variază de la 1bit/pixel la 8 biți/pixel** (imagini cu nuanțe de gri). O rezoluție mai mare nu este necesară deoarece ochiul uman nu percepe mai multe nuanțe de gri. **Pentru imaginile color se alocă separat biți pentru cele 3 culori de bază (roşu, verde şi albastru)** sau pentru tripletul culoare / luminanță / intensitate.

Pentru imagini în mişcare (film) se achiziționează cadre (imagini statice) cu o frecvență de eşantionare cuprinsă între 10Hz (film de slabă calitate, transmisibilă pe o legătură Internet de mică viteză) *şi 50-70Hz* (film de calitate TV sau chiar mai bună) [Seb, Gor05].

În aceste condiții memorarea unei informații video de 10 secunde la o rezoluție şi calitate acceptabilă necesită un spațiu de memorie de: (3 octeți/pixel)*(1000 pixeli/linie)*(700 linii/cadru)*(50 cadre/sec.)*10 secunde = 1.050.000.000 octeți =1Goctet. Este de remarcat faptul că **calculatoare personale actualele nu dispun de o memorie internă care să memoreze o astfel de informație** (un film de 100 de minute necesită ≈600 Gocteți). De aceea şi în cazul informațiilor video se impune utilizarea unor metode eficiente de compactare, care să reducă spațiul necesar de memorare. Majoritatea metodelor de compactare sunt cu pierdere de informație; pentru un factor de compactare mai mare se face un compromis în ceea ce priveşte calitatea imaginii.

2.2. CODIFICAREA INFORMAȚIEI

Se numeşte **cod**, un set de *simboluri elementare* împreună cu o serie de reguli potrivit cărora se formează aceste simboluri [Mâr96]. **Codificarea** reprezintă procesul de stabilire a unui cod. Dacă se notează prin $X=\{x_1, x_2, ..., x_n\}$ mulțimea datelor în forma accesibilă utilizatorului, iar cu $Y=\{y_1, y_2, ..., y_m\}$ mulțimea simbolurilor interpretate de sistemul de calcul, atunci codificarea reprezintă asocierea fiecărui element $x_i \in X$ unei secvențe de simboluri $y_i \in Y$; în acest caz, codul este o funcție bijectivă $f : X \rightarrow Y$. Funcția inversă $f^{-1} : Y \rightarrow X$ reprezintă procesul de decodificare a secvențelor de simboluri utilizate de sistemul de calcul într-o formă înțeleasă de utilizator.

Obiectivele unui sistem de codificare sunt:

1. *reprezentarea informațiilor într-o formă cât mai simplă*
2. *facilitarea implementării operațiilor aritmetice şi logice*
3. *detecția şi corecția erorilor* (paritate, CRC, suma de control)
4. utilizarea unui spațiu minim de memorare (*compactarea informației*)
5. protecția împotriva accesului neautorizat (*securitate*)

2.2.1. CODURI ZECIMALE

Pentru anumite aplicații este important ca datele numerice să se păstreze în formă zecimală. În acest fel pot fi evitate conversiile repetate din zecimal în binar şi invers. O posibilitate este utilizarea a 4 biți pentru a reprezenta cele 10 cifre zecimale (0, 1, ... 9). Reprezentarea poartă numele de codul BCD – Binary Coded Decimal. Acest cod este asemănător cu reprezentarea hexazecimală cu diferența că se utilizează numai primele 10 combinații de biți:

0_{10} -> 0000_2, 1_{10} -> 0001_2, 2_{10} -> 0010_2, ... 9_{10} -> 1001_2 ; celelalte combinații sunt nepermise (ex. 1010, ... 1111)

Procesoarele Intel au instrucțiuni în limbaj de asamblare care suportă operații aritmetice în reprezentarea BCD. Codurile cifrelor zecimale se păstrează fie individual pe câte un octet (forma despachetată), fie câte 2 cifre pe un octet (forma împachetată).

2.2.2. CODURI PENTRU DETECTAREA ŞI CORECTAREA ERORILOR

Există mai multe modalități de **detectare a erorilor** în cadrul transmisiei, bazate în general pe ataşarea unor cifre binare de control la emisia mesajului prezentat sub formă binară; recepția va controla modul de respectare a corectitudinii mesajului. Cele mai utilizate procedee sunt [Mâr96]:

- **Codurile de control a parității**: la emisia unei succesiuni de cifre binare a_1, a_2, …, a_n se ataşează o cifră binară de control a_{n+1} aleasă astfel încât numărul total de cifre binare de 1 să fie *par* sau *impar*, în funcție de convenția de paritate stabilită. Are o eficiență scăzută întrucât detectează numai erorile care au un număr impar de biți modificați (1, 3,...).
- Detecția bazată pe **distanță Hamming** (d – numărul minim de biți prin care diferă două coduri valide.
- **Codurile polinomial ciclice (CRC) –** este metoda cea mai folosită pentru detectarea erorilor grupate. Înaintea transmiterii, informației i se adaugă biți de control, iar pe baza acestora, la dacă la recepționarea mesajului se detectează erori, atunci acesta trebuie retransmis. O informație pe *n* biți poate fi considerată ca lista coeficienților binari ai unui polinom cu *n* termeni, deci de grad *n-1*.

  Exemplu: $110001 \rightarrow x^5 + x^4 + 1$

  Pentru a calcula biții de control se va efectua un anumit număr de operații cu aceste polinoame cu coeficienți binari. Operațiile se vor efectua *modulo 2*, adunarea şi scăderea nu va ține seama de cifra de transport, deci toate operațiile de adunare şi scădere sunt identice cu operația XOR. Pentru generarea şi verificarea biților de control atât sursa cât şi destinația mesajului utilizează un polinom generator.

Există şi coduri care după detectarea erorilor oferă posibilitatea **corectării** unei singure cifre binare detectată incorectă. Dintre cele mai frecvent utilizate se pot menționa:

- **Codurile de paritate încrucişată.** În acest caz succesiunea de cifre binare este divizată în secvențe binare de aceeaşi lungime. La emisie, fiecărei linii şi coloane li se ataşează o cifră de control conform parității stabilite. La recepție, prin controlul parității pe fiecare linie şi coloană, se detectează eventualele erori având posibilitatea de a corecta o eroare apărută la intersecția liniei şi coloanei detectate ca nerespectând paritatea.

- **Codul Hamming 4+3** (ataşează la emisie pentru fiecare 4 biți de date şi 3 biți de control) asigură detecția poziției bitului eronat şi implicit permite corectarea acestuia (prin complementarea poziției eronate). Structura codului Hamming este:

c_1	c_2	a_3	c_4	a_5	a_6	a_7

unde,

a_3, $a_5 \div a_7$ este mesajul (datele) de transmis

c_1, c_2, c_4 reprezintă cifrele de control, care se determină astfel:

$c_1 = a_3 + a_5 + a_7$

$c_2 = a_3 + a_6 + a_7$

$c_4 = a_5 + a_6 + a_7$

Se observă că transmisia prin codul Hamming mărește secvența binară ce se transmite cu 75%, dar oferă un control mult mai riguros. La recepția secvențelor binare, la fiecare grup de 7 cifre se procedează astfel:

- Se calculează sumele:

$S_0 = c_1 + a_3 + a_5 + a_7 = 2 \times c_1$

$S_1 = c_2 + a_3 + a_6 + a_7 = 2 \times c_2$

$S_2 = c_4 + a_5 + a_6 + a_7 = 2 \times c_4$

- Se ponderează fiecare sumă cu 2^0, 2^1, 2^2:

$n_0 = 2^0 \times S_0$

$n_1 = 2^1 \times S_1$

$n_2 = 2^2 \times S_2$

- Se calculează $n_0 + n_1 + n_2$; dacă această sumă este 0 atunci mesajul recepționat este corect, în caz contrar aceasta reprezintă poziția care este eronată.

2.3. OPERAȚII LOGICE PE BIȚI. FUNCȚII LOGICE.

2.3.1. FUNCȚIA ŞI LOGIC (*AND*)

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

2.3.2. FUNCȚIA SAU LOGIC (*OR*)

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

2.3.3. FUNCȚIA NOT LOGIC

A	NOT
0	1
1	0

2.3.4. FUNCȚIA SAU EXCLUSIV (*XOR*)

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Regulile lui De Morgan

1. NOT (**A** AND **B**) = (NOT **A**) OR (NOT **B**)
2. NOT (**A** OR **B**) = (NOT **A**) AND (NOT **B**)

Figura 2.1 încearcă să dea o interpretare grafică (bazată pe mulțimi şi pe operațiile dintre acestea – reuniune, intersecție, diferență) a primei reguli a lui De Morgan. Cercul mare reprezintă mulțimea tuturor cazurilor posibile.

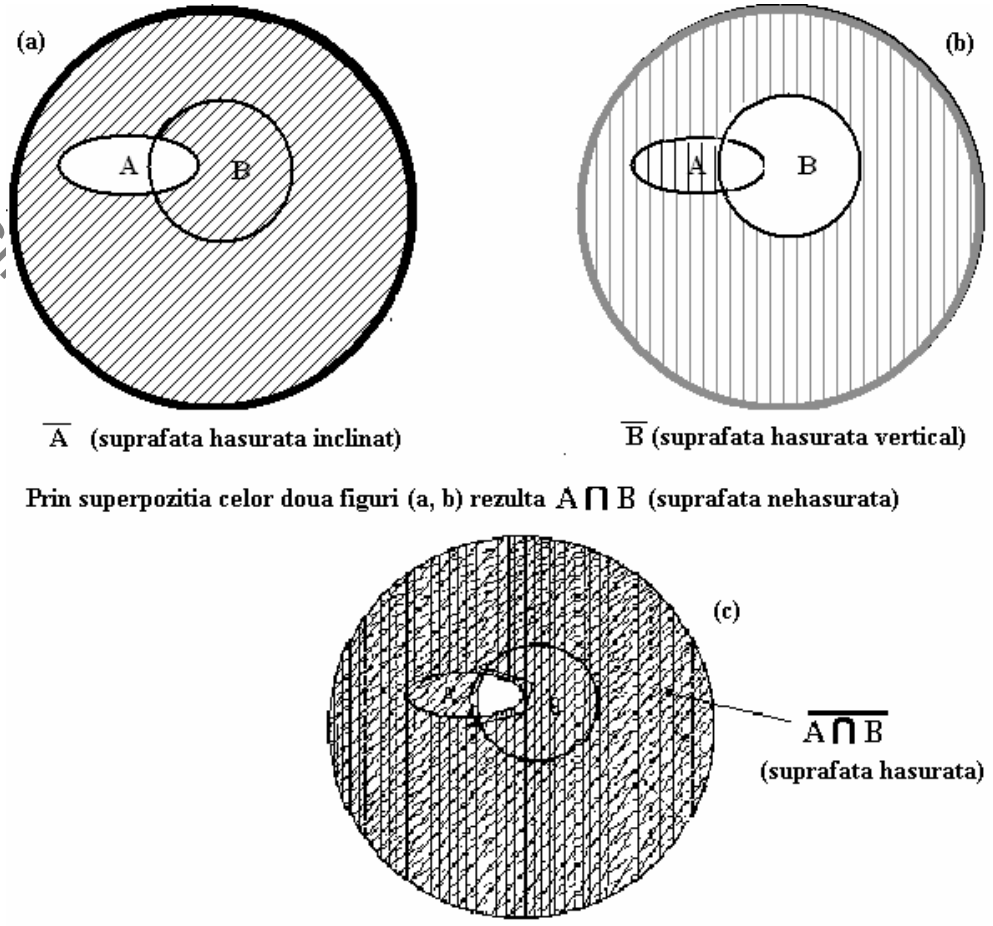


Figura 2.1. **Verificarea regulii lui De MORGAN** ***NOT (A AND B) = (NOT A) OR (NOT B)***

2.4. EXERCIȚII ŞI PROBLEME

1. Să se convertească următoarele numere din baza 10 în baza 2 şi în baza 16:

0, 1, 1243, 256, 2048, 5655, 123.14, 33.125, 25675, 675.625

2. Să se convertească următoarele din baza 2 în baza 10 şi 16:

110100001100, 101.111011, 110011.11

Să se exprime în hexazecimal următoarele şiruri de caractere:
ABCDEF, hello. Arătaţi cum vor fi stocate în memorie aceste şiruri ştiind că procesorul Intel foloseşte convenţia *little endian*.

3. Să se convertească următoarele numere din hexazecimal în binar:

AFE, ABC1, 12BD, E000,

4. Să se evalueze numărul de biţi necesari pentru a reprezenta numere întregi în intervalele:

0-100, 0-1000, 300-400, -100 ÷ +100

5. Să se reprezinte următoarele numere în semn şi mărime, complement faţă de 1 şi complement faţă de 2, pe 8 biţi:

0, 13, -100, -44, -1, -130

6. Să se precizeze domeniul maxim de valori ce se poate reprezenta pe un octet şi pe 2 octeţi dacă se consideră reprezentarea numerelor strict pozitive şi respectiv a numerelor cu semn.

7. Să se reprezinte următoarele numere în virgulă flotantă simplă precizie:

1, 0.000023, 100000.000001, 13,4

Să se comenteze limitările reprezentării în virgulă flotantă: domeniul maxim de valori, valoarea minimă reprezentabilă în valoare absolută, pasul minim între două valori mari consecutive, etc.

8. Să se descrie în pseudocod câteva tehnici de compactare a informaţiei (ex: codificarea secvenţelor lungi de aceeaşi valoare, codificarea variaţiilor între eşantioane consecutive, etc.)

9. Se vor vizualiza date din memoria şi registrele calculatorului, folosind un program de depanare (SPIMSal, TD, LC2, etc.).

10. Se va evalua necesarul de spaţiu de memorie pentru diferite tipuri de aplicaţii (aplicaţii care prelucrează informaţii numerice, grafică, alfanumerice sau multimedia).

11. Probleme de conversie dintr-o baza in alta: Scrieţi un program care converteşte numere întregi în şi din binar, zecimal şi hexazecimal. Programul poate fi scris fie în Java, C++ sau C şi vor exista trei argumente în linia de comandă care apelează executabilul realizat.

a) Un caracter care simbolizează din ce sistem de numerație se face conversia: - *h* pentru hexazecimal, *d* pentru zecimal sau *b* pentru binar.
b) Un alt caracter care exprimă sistemul de numerație destinație: *h*, *d* sau *b*.
c) Un şir de digiți (cifrele 0÷9 şi eventual literele A÷F) reprezentând numărul supus conversiei

Programul va afişa pe ecran (ieşirea standard a sistemului) unul din următoarele mesaje:

- Dacă argumentele sunt date corect, se afişează un mesaj care conține: sistemul de numerație sursă, numărul inițial în acest sistem, sistemul de numerație destinație şi numărul în sistemul de numerație final (De exemplu: `16 în decimal este 10000 în binar`).
- Altfel, se afişează un mesaj de ajutor pentru utilizarea parametrilor de comandă ai programului (De exemplu: Folosiți `java hex [ h | d | b ] [ h | d | b]`.

Considerând că programul este scris în *java* şi numit *hex.java* se dau în continuare câteva secvențe de apel cu rezultatele obținute.

```
% java hex b d 1010
1010 în binar este 10 în zecimal

% java hex h d 123456
123456 în hexazecimal este 1193046 în zecimal

% java hex d h 12345678
12345678 în zecimal este bc614e în hexazecimal

% java hex a b 123
Folosiți java hex [ h | d | b ] [ h | d | b]

% java hex h d bc614e
bc614e în hexazecimal este 12345678 în zecimal
```

12. Scrieți un algoritm eficient din punct de vedere al reprezentării în memorie pentru rezolvarea problemei 2^n cu n foarte mare, n≥100. Determinați numărul de digiți pe care se reprezintă numărul rezultat. Ca şi indicație, se poate folosi aproximarea grosieră $2^{10}\approx1000$.

3. STRUCTURI LOGICE DIGITALE. TRANZISTORI. PORȚI LOGICE. STRUCTURI LOGICE COMBINAȚIONALE. UNITATEA ARITMETICO-LOGICĂ

3.1. TRANZISTORUL

Tranzistorul este un dispozitiv electronic, semiconductor, inventat la laboratorul de telefonie Bell din New Jersey în decembrie 1947 de către cercetătorii în fizică John Bardeen, Walter Houser Brattain, şi directorul celor doi, William Bradford Shockley [Wik]. Pentru invenția lor cei trei au fost recompensați în 1956 cu premiul Nobel. Descoperirea tranzistorului a determinat dezvoltarea electronicii, acesta fiind considerat una din cele mai mari descoperiri ale erei moderne. Tranzistorul poate fi folosit atât în domeniul analogic pentru a amplifica, stabiliza sau modula diverse semnale electrice cât şi în cel digital pentru comutare.

Din punct de vedere al complexității microprocesoarelor se poate spune că numărul tranzistorilor integrați într-un cip se dublează la fiecare 18 luni. Afirmația aparține lui Gordon Moore şi datează de la mijlocul anilor ’60. După cum se poate observa din următorul tabel afirmația este pe deplin justificată de implementările comerciale [Katz05].

Anul lansării pe piață	**Denumire procesor**	**Număr de tranzistori încorporați / Tehnologia de integrare**	**Frecvența procesorului**	**Magistrala de date**
1971	Intel 4004	2300 / 10µm	108KHz	4-biți
1972	Intel 8008	3500 / 10µm	200KHz	8-biți
1974	Intel 8080	6000 / 6µm	2MHz	8-biți
1978	Intel 8086 / 8088	29000 / 3µm	4.77-10MHz	16-biți. Implementat în primul

				calculator personal IBM.
1982	Intel 80286	134000 / 1.5µm	12.5-20MHz	16-biți
1985	Intel 80386DX	275,000 / 1µm	33MHz	32-biți
1989	Intel 80486DX	1200000 / 0.8µm	25-100MHz	32-biți. Primul procesor care încorporează o memorie de tip cache.
1993	Intel Pentium	3100000/ 0.5µm	60-133MHz	32-biți
1995	Intel Pentium Pro	5500000 / 0.35µm	200MHz	32-biți
1997	Intel Pentium II	7500000 / 0.35µm	233-333MHz	32-biți
2000	Intel Pentium III	28000000 / 0.18µm	733MHz-1.2GHz	32-biți
2001	Intel Pentium 4 ver. Northwood	42000000 / 0.18µm	1.6 GHz	32-biți (din 2000 apare versiunea Itanium pe 64 de biți)
2003	Intel Pentium 4 ver. Northwood Hyperthread	55000000 / 0.13µm	2 – 3.4 GHz	32-biți
2004	Intel Pentium 4 ver. Extrem Edition Hyperthread	178000000 / 0.09µm	2,8 - 3,4GHz	32-biți

Tabelul 3.1 Procesorul Intel – tendințe tehnologice

Referitor la tabelul 3.1 sunt câțiva parametri care vor fi înțeleşi mai bine după parcurgerea acestui capitol.

- Dimensiunea tehnologiei de integrare reprezintă o caracteristică importantă în evaluarea şi creşterea performanţei arhitecturilor de calcul şi implicit al memoriilor de tip cache. Exprimă dimensiunea unei porţi logice (diferenţa în microni dintre ieşirea şi intrarea unei porţi logice fundamentale NAND). În unele documentaţii este interpretată ca fiind *distanţa dintre două intrări aferente porţii logice.*

Ca şi corolar al tabelului 3.1 se poate afirma că şi frecvenţa procesoarelor pare să se dubleze în acelaşi ritm cu toate că, creşterea în viteză se datorează tendinţelor tehnologice doar într-o proporţie de 35%, restul de 65% rezultând în urma îmbunătăţirilor arhitecturale [Flo05].

Tranzistorii sunt construite din circuite (în tehnologie) **MOS (metal oxid semiconductor)** [Patt03]. Există două tipuri de tranzistori (vezi figura 3.1): de tip **P** (pozitiv) şi de tip **N** (negativ). Au funcţii complementare, astfel încât folosite împreună se pot realiza circuite **CMOS**. Tranzistorul are rol de întrerupător: **închide circuitul** (valoarea tensiunii din A este egală cu cea din B) sau **deschide circuitul** (valorile celor două tensiuni din A şi B diferă – legătura fiind întreruptă).

Funcţionarea celor două tipuri de tranzistori este descrisă în cadrul figurii 3.1.

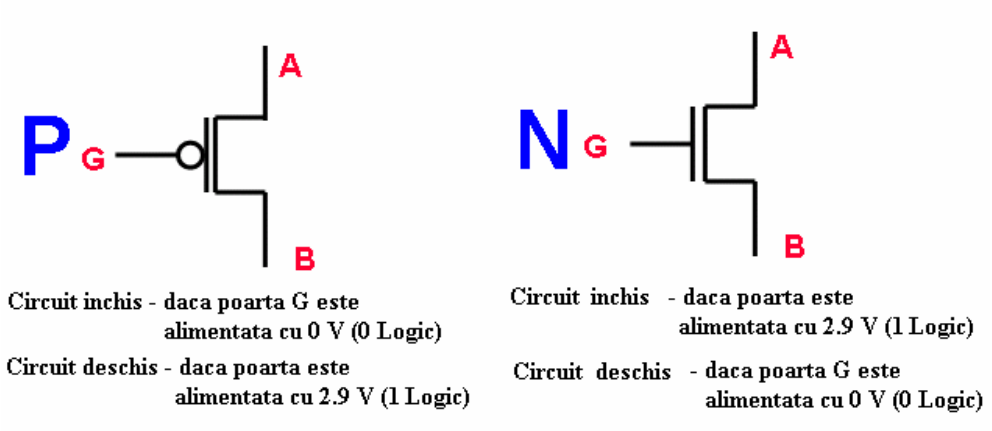


Figura 3.1. Tranzistori de tip P şi de tip N

3.2. PORȚI LOGICE

3.2.1. INVERSORUL CMOS

- Dacă IN = 0 (adică 0 V) atunci tranzistorul de tip P va conduce iar cel de tip N rămâne blocat. Rezultă că ieşirea OUT va avea 2.9 V (adică 1 Logic).
- Dacă IN = 1 (adică 2.9 V) atunci tranzistorul de tip N va conduce iar cel de tip P rămâne blocat iar ieşirea OUT va avea 0 V (adică 0 Logic).

Se poate astfel observa că OUT = Not (IN); ieşirea este inversa intrării.

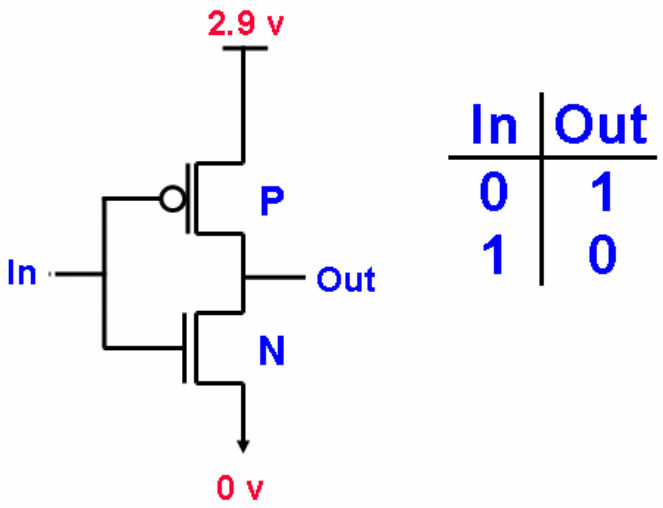
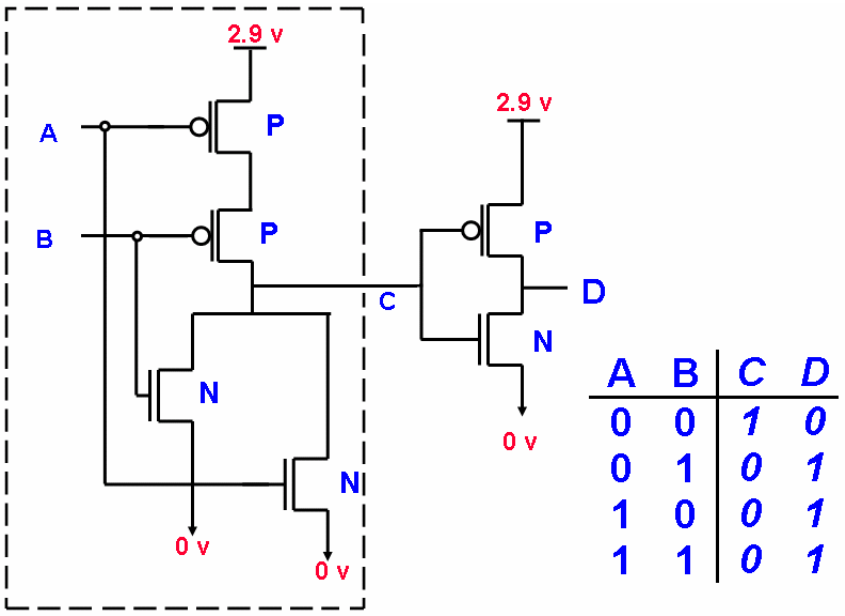


Figura 3.2. Inversorul CMOS

3.2.2. POARTA NOR (*NOT OR*)

Poarta NOR este realizată din doi tranzistori de tip P legați în serie şi doi tranzistori de tip N aşezați în paralel (vezi figura 3.3 – zona încadrată cu linie punctată). Astfel, pentru a avea la ieşire în punctul C 2.9V trebuie ca cei doi tranzistori de tip P să conducă (ambele circuite închise); deci A=0 ŞI B=0 $\Rightarrow$ C=1.

Pentru a avea la ieşire în C 0V este suficient ca măcar unul din cei doi tranzistori de tip N să conducă (cel puţin un circuit închis); deci A=1 SAU B=1 ⇒ C=0. Prin utilizarea unui inversor la ieşirea porţii NOR, în punctul D, se obţine rezultatul funcţiei OR aplicat celor două intrări A şi B.



A	B	C	D
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

Figura 3.3. Porţile NOR şi (OR)

3.2.3. POARTA NAND (*NOT AND*)

Poarta NAND este implementată din doi tranzistori de tip P aşezaţii în paralel şi doi tranzistori de tip N în serie (vezi figura 3.4 – zona încadrată cu linie punctată). Astfel, pentru ca la ieşirea C să fie 2.9V trebuie ca cel puţin unul din cei doi tranzistori de tip P să conducă (cel puţin un circuit închis); deci A=0 SAU B=0 ⇒ C=1.

Pentru a avea la ieşire în C 0V trebuie ca ambii tranzistori de tip N să conducă (ambele circuite închise); deci A=1 ŞI B=1 ⇒ C=0. Prin folosirea inversorului la ieşirea porţii NAND, în punctul D, se obţine rezultatul funcţiei AND aplicat celor două intrări A şi B.

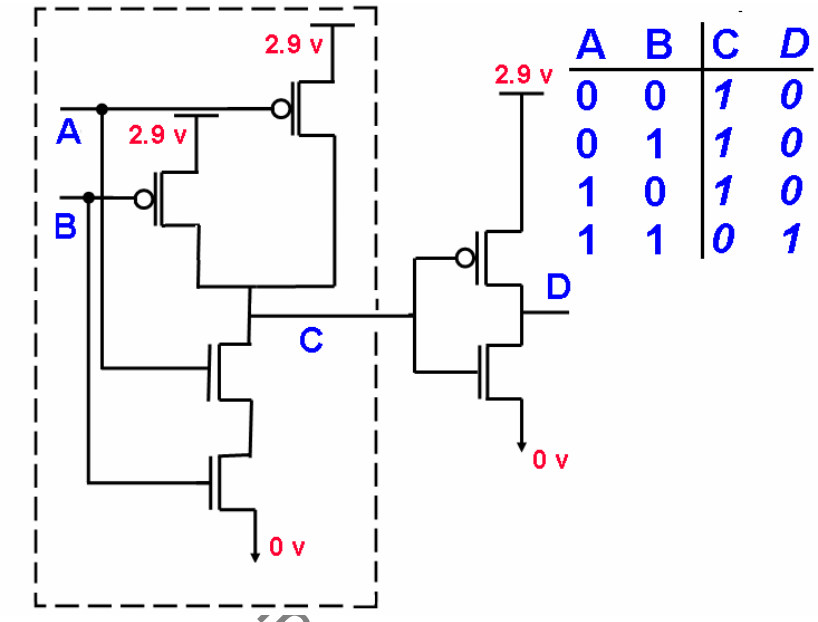


Figura 3.4. Porțile NAND şi (AND)

Întrucât pentru exprimarea funcțiilor logice de 2 operanzi (AND, OR) sunt necesari 6 tranzistori şi ținând cont că la nivelul procesoarelor anilor 2004 [Katz05] erau implementați 178 milioane de astfel de circuite digitale primare, rezultă necesitatea unei convenții de reprezentare în vederea simplificării procesului de proiectare. Trebuie specificat că porțile logice (AND, OR, NAND, NOR) pot avea mai mult de două intrări.

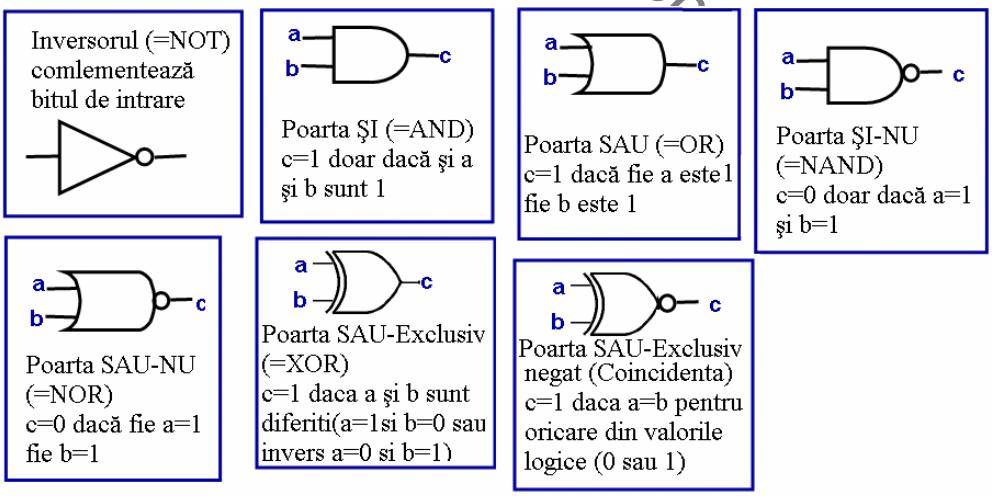


Figura 3.5. Convenții de reprezentare a porților logice

Reprezentarea funcțiilor logice se poate face prin:

- Tabel de adevăr
- Expresie logică
- Circuite logice

Se spune despre setul de porți logice (AND, OR, NOT) că este **complet** întrucât cele trei porți logice sunt suficiente pentru reprezentarea oricărei funcții logice. Făcând legătura cu legile lui De Morgan:

A OR B = NOT ((NOT A) AND (NOT B))

A AND B = NOT ((NOT A) OR (NOT B))

se poate spune că sunt suficiente fie porți de tipul *NAND* fie porți de tipul *NOR*.

Funcțiile ŞI-NU (*NAND*) respectiv SAU-NU (*NOR*) sunt suficiente pentru reprezentarea oricărei funcții logice (se mai numeşte completitudinea funcțiilor logice ŞI-NU şi respectiv SAU-NU) [Pat03].

3.3. STRUCTURI LOGICE DIGITALE

Se cunosc două tipuri de structuri logice digitale:

1. **Structuri de decizie** – pot realiza o decizie dar nu reţin ce fel de decizie a fost (0 sau 1). Se mai numesc *structuri logice combinaționale* deoarece ieşirile acestora sunt strict dependente de combinația valorilor de intrare. Nu depind de istoria informației întrucât aceste structuri nu stochează nici un fel de informație. Se cunosc trei astfel de structuri:
   - **Decodificatorul**
   - **Multiplexorul**
   - **Sumatorul**
2. **Structuri de memorare** – permit stocarea informației. Capacitatea acestora se măsoară în biți sau multiplii ai acestora.

3.3.1. STRUCTURI LOGICE (CIRCUITE) COMBINAȚIONALE

Circuitele logice combinaționale (CLC) fac parte din familia sistemelor automate cu număr finit de stări, fiind un caz particular, acela al

automatelor fără memorie. Lipsa memoriei implică independența circuitului de variabila timp şi deci lipsa unor stări interne. Depinde exclusiv de variabilele aplicate la intrare.

Parametrii:

- .. **Adâncimea** unui circuit este dată de numărul de circuite aflat pe cea mai lungă ramură care compune circuitul. Ea corespunde „***timpului de execuție***" în cazul cel mai defavorabil.
- .. **Dimensiunea** unui circuit reprezintă numărul de elemente combinaționale conținute în circuit.

O întrebare firească ce se poate pune este următoarea: "*Cine determină viteza sau frecvența maximă a unui sistem digital ?*" Răspunsul la această întrebare este: *latența* (sau întârzierea) circuitului. Există câțiva factori care contribuie la această latență [Pop86]. Unul dintre aceştia este timpul de propagare printr-o poartă logică (vezi figura 3.6), un altul îl reprezintă factorul de încărcare al *bufferelor* (nivelului) de ieşire (lungimea anumitor fire de legătură care reprezintă aceeaşi intrare pentru mai multe porți logice) şi chiar circuitul însuşi caracterizat de adâncime şi dimensiune. În realitate, când semnalul de la intrarea unei porți se schimbă (din 0 în 1 sau din 1 în 0) semnalul de ieşire nu se va schimba instantaneu (vezi figura 3.6).

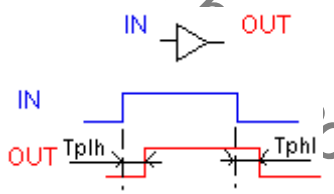


Figura 3.6. Întârzierea semnalului printr-o poartă logică

Timpul de propagare printr-o poartă logică reprezintă diferența în timp dintre momentele în care este modificat semnalul pe intrarea porții logice şi respectiv momentul în care este sesizată modificarea semnalului la ieşirea porții logice. Se disting două tipuri de întârzieri prin poarta logică T_{PHL} (timp de propagare din nivel logic High în nivel logic Low) şi respectiv T_{PLH} (vezi figura anterioară). Valoarea timpului de întârziere variază în funcție de poarta logică folosită precum şi în funcție de familia de circuite logice folosite. În general, cu cât prețul per cip este mai mare cu atât acesta va fi mai rapid. La nivelul anului 2004, timpul de întârziere printr-o poartă logică era cuprins în intervalul 1.5-4ns.

3.3.1.1. DECODIFICATORUL

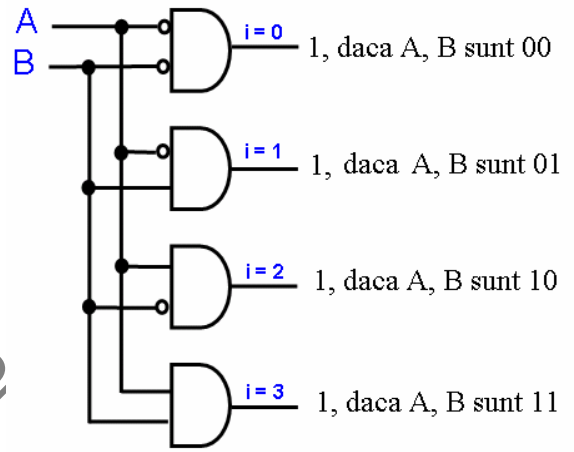


Figura 3.7. Decodificatorul - reprezentare cu porţi logice

În general au *n* intrări şi 2^n ieşiri. Decodificatorul furnizează la o singură ieşire 1 iar la restul 0. Ieşirea „*i*" va avea valoarea logică 1 dacă *numărul reprezentat în binar pe intrare* este chiar 1. Funcţia decodificatorului este de interpretare a unui *pattern* de biţi. Utilitatea sa se regăseşte în faza de *decodificare* a instrucţiunilor (vezi modelul *von Neumann*) când câmpul *opcode* – parte componentă a oricărei instrucţiuni – va specifica ce operaţie trebuie efectuată (adunare / scădere / înmulţire / împărţire / acces la memorie / ramificaţie a programului). Decodificatorul este utilizat şi pentru identificarea regiştrilor sursă / destinaţie sau a modurilor de adresare în cazul instrucţiunilor cu referire la memorie tot pe baza *pattern*-urilor de biţi din corpul instrucţiunilor. De asemenea, este folosit la selecţia adreselor de memorie.

3.3.1.2. MULTIPLEXORUL

Are rolul de a selecta o intrare şi de a o conecta la ieşire. Semnalul de selecţie (codificarea binară a acestuia) determină care dintre intrări va fi conectată la ieşire.

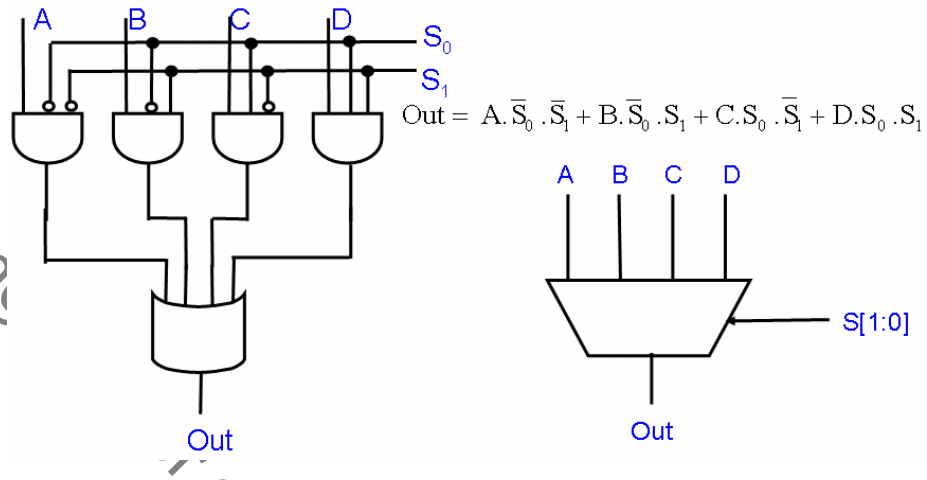


Figura 3.8. Multiplexor 4:1 - reprezentare cu porți logice

În general, un MUX constă din 2^n intrări, *n* linii de selecție şi o singură ieşire. Multiplexorul este utilizat în proiectarea procesoarelor pipeline în cel puțin trei ipostaze:

- **La selecția adresei următoarei instrucțiuni** dintre *PC+4* (o procesare secvențială, instrucțiunea fiind codificată pe 4 octeți – cazul procesorului MIPS R3000 [Flo03]) şi *PC+adresa_relativă* (identificarea unei ramificații în program) în funcție de bitul rezultat (se face / nu se face).
- **La selecția operanzilor unei instrucțiuni:** pot fi instrucțiuni cu 2 regiştri sursă, pot fi instrucțiuni aritmetico-logice cu un registru sursă şi o valoarea imediată, poate fi un calcul de adresă dintre un registru index şi un deplasament (mod de adresare indexat), poate fi calculul adresei destinație în cazul unei ramificații de program (registru + valoare imediată). Pentru detalii vezi subcapitolul 6.2.
- **Scrierea rezultatului instrucțiunilor în setul de regiştri generali ai procesorului**, în faza *store results* de procesare (vezi modelul *von Neumann*) poate fi făcută prin instrucțiuni *aritmetico-logice* sau instrucțiuni de tip *load*. Selecția se va face tot intermediul unui multiplexor 2:1 comandat cu un bit din opcod-ul instrucțiunii.

3.3.1.3. SUMATORUL

Efectuează operații aritmetice (adunare sau scădere) cu două numere binare având un număr egal de biți. Orice sumator pe mai mulți biți este construit din sumatoare elementare pe un bit.

Sumatoarele elementare pe un bit pot fi:

- *semisumatoare* (sumator pentru bitul zero), acest sumator elementar se caracterizează prin faptul că nu ține seama de transportul de la bitul cu semnificație imediat inferioară (vezi figura 3.9).
- *sumatoare complete pe un bit* care ţin seama de transportul de la bitul cu semnificație imediat inferioară.

Se cunosc trei circuite combinaționale de însumare. Primul este **sumatorul cu transport propagat**, care poate să adune două numere de câte *n* biți în timp $\theta(n)$ şi utilizând un circuit de dimensiune $\theta(n)$ [Cor90]. Al doilea este **sumatorul cu transport anticipat**, de dimensiune $\theta(n)$ care adună în timp $\theta(\log_2 n)$. Al treilea este un **circuit cu transport salvat**, care, în timp $\theta(1)$ (constant), poate reduce suma a trei numere de câte *n* biți la suma unui număr de *n* biți şi a unuia de *n*+1 biți. Circuitul are dimensiunea $\theta(n)$.

Semisumatorul (sumatorul pentru bitul zero) [Patt03, Clc04]

- intrările celor două numere pe un bit sunt reprezentate prin X_0 şi Y_0.
- ieşirile sunt: S_0 - (**suma** celor două numere) şi - (*Carry* - **transportul** către bitul 1).

Funcționarea semisumatorului

X_0	Y_0	C_1	S_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

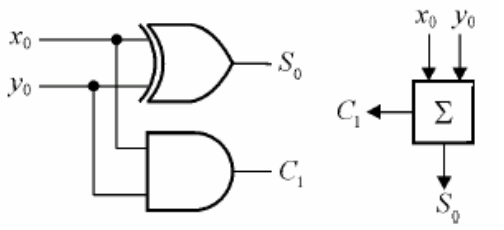


Figura 3.9. Semisumator pe un bit

Sumatorul complet pe un bit

Sumatorul complet pe un bit ține cont de transportul de la bitul de semnificație imediat inferioară. Are intrările: X_n, Y_n, C_n şi ieşirile: S_n, C_{n+1}. Funcționarea sa se bazează pe tabelul de mai jos. Din tabel se deduc relațiile (3.1) şi (3.2) care descriu dependența ieşirilor de intrări:

$$S_n = \overline{X_n} \cdot \overline{Y_n} \cdot C_n + \overline{X_n} \cdot Y_n \cdot \overline{C}_n + X_n \cdot \overline{Y_n} \cdot \overline{C}_n + X_n \cdot Y_n \cdot C_n = C_n \cdot (\overline{X_n} \cdot \overline{Y_n} + X_n \cdot Y_n) + \overline{C}_n \cdot (\overline{X_n} \cdot Y_n + X_n \cdot \overline{Y_n}) =$$
$$= C_n \cdot \overline{X_n \oplus Y_n} + \overline{C}_n \cdot X_n \oplus Y_n = C_n \oplus X_n \oplus Y_n \quad (3.1)$$

$$C_{n+1} = \overline{X_n} \cdot Y_n \cdot C_n + X_n \cdot \overline{Y_n} \cdot C_n + X_n \cdot Y_n \cdot \overline{C}_n + X_n \cdot Y_n \cdot C_n = X_n \cdot Y_n \cdot (C_n + \overline{C}_n) + C_n \cdot (\overline{X_n} \cdot Y_n + X_n \cdot \overline{Y_n}) =$$
$$= X_n \cdot Y_n + C_n \cdot X_n \oplus Y_n \quad (3.2)$$

Funcționarea sumatorului complet

X_n	Y_n	C_n	C_{n+1}	S_n
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

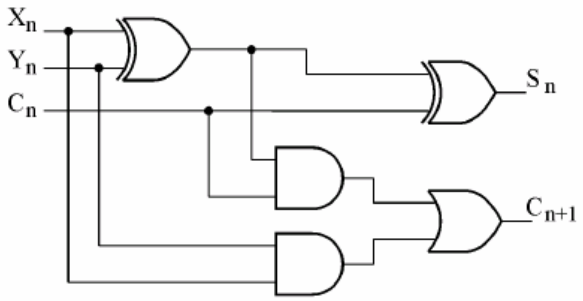


Figura 3.10. Sumator complet pe un bit – varianta (1)

Din relația (3.2) se deduce următoarea schema din figura 3.11, mai rapidă (scade numărul de nivele cu 1 – o poartă):

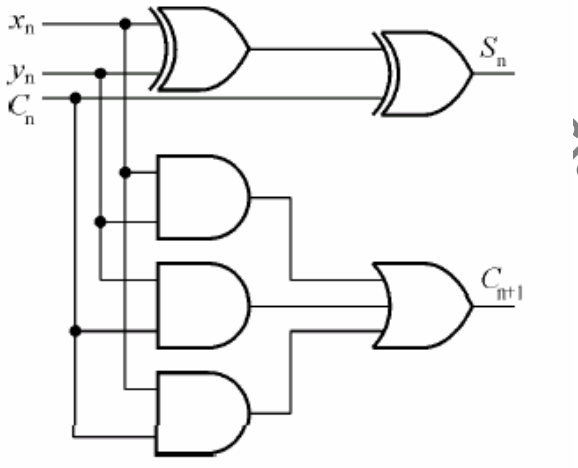


Figura 3.11. Sumator complet pe un bit – varianta (2)

$$C_{n+1} = \overline{X_n} \cdot Y_n \cdot C_n + X_n \cdot \overline{Y_n} \cdot C_n + X_n \cdot Y_n \cdot \overline{C_n} + X_n \cdot Y_n \cdot C_n = \overline{X_n} \cdot Y_n \cdot C_n + X_n \cdot \overline{Y_n} \cdot C_n + X_n \cdot Y_n \cdot \overline{C_n} + X_n \cdot Y_n \cdot C_n +$$
$$+ X_n \cdot Y_n \cdot C_n + X_n \cdot Y_n \cdot C_n = Y_n \cdot C_n \cdot (X_n + \overline{X_n}) + X_n \cdot C_n \cdot (Y_n + \overline{Y_n}) + X_n \cdot Y_n \cdot (C_n + \overline{C_n}) = Y_n \cdot C_n + X_n \cdot C_n + X_n \cdot Y_n \quad (3.3)$$

Pentru deducerea relației (3.3) am folosit faptul că X + X = X, pentru $\forall X \in \{0, 1\}$.

Sumator cu transport succesiv (propagat) 74LS83 (4 biți) [Clc04]

Schema acestui sumator pe patru biți cuprinde patru sumatoare complete pe un bit interconectate ca în figura 3.12:

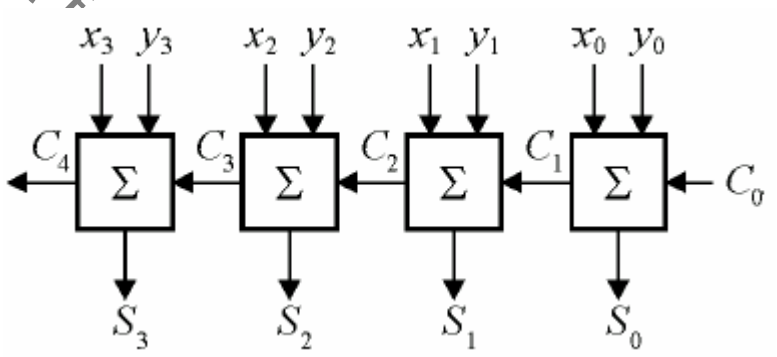


Figura 3.12. Sumator cu transport succesiv pe 4 biți (circuitul 74LS83)

C_0 se pune la masă dacă circuitul este folosit pentru însumarea a două numere cu 4 biți, deoarece nu există transport de la un bit cu semnificație mai mică. Când se extinde numărul de biți folosind două sau mai multe circuite conectate se face concordanța cu schema din figura 3.13:

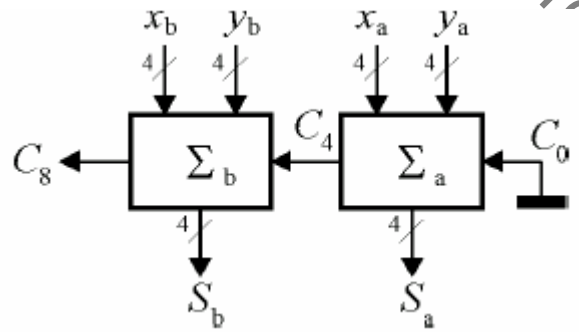


Figura 3.13. Extinderea capacității de adunare

Considerând *t* întârzierea pe o poartă logică elementară (sau pe un nivel logic), observând schema din figura 3.11, rezultă că C_1 se obține cu o

întârziere de $2 \cdot t$. Întârzierea cu C_{n+1} (transportul obținut la ieşirea ultimului sumator – de rang n) este $2 \cdot t$ $(n+1)$, ceea ce este inacceptabil pentru sumatoare cu număr mare de ranguri. Timpii de calcul pentru biții de sumă ($S_0 \div S_n$) sunt:

$t_{S1} = 2 \cdot t$.

$t_{S2} = t + t_{C1} = \cdot t + \cdot 2 \cdot t = 3 \cdot t$

...

$t_{Sn} = t + t_{Cn-1} = t + \cdot 2 \cdot (n-1) \cdot t = (2 \cdot n-1) \cdot t$

Îmbunătățirile au dublu scop:

- simplificarea structurii sumatorului
- reducerea timpului de însumare.

Sumator cu întârziere minimă

Sumatorul cu transport succesiv are o structură simplă dar prezintă marele dezavantaj că orice operație de însumare necesită un timp foarte mare pentru execuție datorită propagării transportului. Imediat ce progresele tehnologice au permis (apariția circuitelor integrate pe scară medie, largă şi foarte largă – MSI, LSI, VLSI) s-a renunțat la această structură integrându-se în circuite specializate, scheme mult mai performante. Sumatorul cu întârziere minimă are la bază ideea că orice funcție de comutație (logică) se poate realiza sub o formă disjunctivă (disjuncție de conjuncții). Relația (3.3), particularizată pentru diverse ranguri de sumare, devine:

$C_1 = X_0 \cdot Y_0 + C_0 \cdot (X_0 + Y_0)$

$C_2 = X_1 \cdot Y_1 + C_1 \cdot (X_1 + Y_1) = X_1 \cdot Y_1 + (X_1 + Y_1) \cdot (X_0 \cdot Y_0 + C_0 \cdot (X_0 + Y_0))$

…

$C_{n+1} = \mathbf{f}(X_{0..n,} \cdot Y_{0..n,} C_0)$ (3.4)

Pe baza relației (3.4) se poate implementa sumatorul cu întârziere minimă. La acest sumator C_{n+1} se determină cu o întârziere de $2 \cdot t$ deoarece depinde doar de $X_{0..n,} \cdot Y_{0..n}$ şi C_0. Dezavantajul constă în numărul mare de porți necesare pentru implementare. Compromisul constă aici în implementarea unor scheme mixte care conțin sumatoare cu întârziere minimă pe grupe de biți şi cu transport propagat între aceste grupe.

Sumator cu transport anticipat

Întrucât sumatorul cu transport succesiv este lent, iar sumatorul cu întârziere minimă este imposibil de realizat, se caută o soluţie intermediară. Aceasta grupează relaţiile obţinute la sumatorul cu întârziere minimă, astfel încât să se obţină dimensiuni rezonabile. Ideea de bază este aceea de a caracteriza comportarea unui rang al sumatorului din punctul de vedere al *generări*i / *propagării* unui transport [Pop86].

La acest tip de sumator se va micşora timpul de execuţie al sumării dar va creşte şi dimensiunea circuitului. Pentru implementare se va pleca de la ecuaţiile ce descriu sumatorul complet (3.1) şi (3.3).

$$S_n = C_n \oplus X_n \oplus Y_n \tag{3.1}$$

$$C_{n+1} = Y_n \cdot C_n + X_n \cdot C_n + X_n \cdot Y_n = X_n \cdot Y_n + C_n \cdot (X_n + Y_n) \tag{3.3}$$

Se consideră notaţia:

$C_{n+1} = G + P \cdot C_n$

unde:

$G = X_n \cdot Y_n$ este termen de generare deoarece pentru $X_n = Y_n = 1$ transportul se generează la nivelul sumatorului complet;

$P = X_n + Y_n$ reprezintă termenul de propagare deoarece o valoare de 1 aplicată numai pe una din intrările sumatorului va permite ca ieşirea C_{n+1} să ia valoarea intrării C_n, care astfel se propagă numai prin acest nivel binar.

Cu aceste notaţii, se va scrie expresia logică a transportului asociat fiecărui ordin binar al unui sumator după cum urmează:

$C_1 = X_0 \cdot Y_0 + C_0 \cdot (X_0 + Y_0) = G_0 + P_0 \cdot C_0$

$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$

$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$

…

$C_i = G_{i-1} + P_{i-1} \cdot G_{i-2} + P_{i-1} \cdot P_{i-2} \cdot G_{i-3} + \ldots + P_{i-1\ldots} \cdot P_1 \cdot G_0 + P_{i-1\ldots} \cdot P_0 \cdot C_0$ (3.5)

…

Propagarea transportului s-a transformat într-un calcul anticipat, în sensul că C_i se poate calcula în paralel pentru i=1, 2, ..., n fără a se mai aştepta calculul mărimilor pentru indicii anteriori i-1,...,1.

3.3.2. PROIECTAREA UNEI UNITĂȚI DE CALCUL ARITMETICO-LOGIC

Unitatea aritmetico-logică (ALU) reprezintă componenta microprocesorului care tratează (execută) toate operațiile aritmetice și logice. ALU este un circuit logic combinațional care efectuează operații aritmetice și logice asupra unei perechi de operanzi pe *n* biți.

Ca și exercițiu se va prezenta în continuare modul de funcționare a unei unități ALU simplificate pe 4 biți (A[3:0] și B[3:0] – vezi schema bloc din figura următoare).

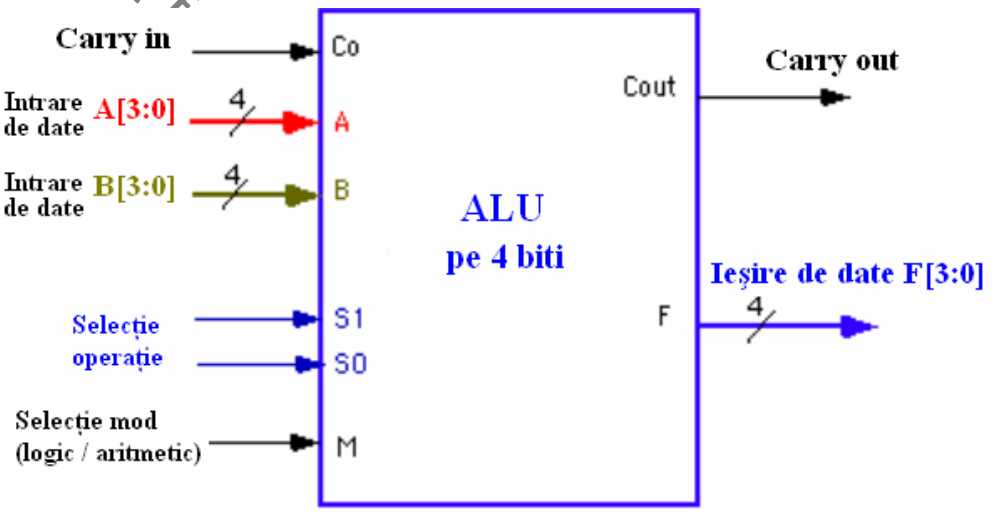


Figura 3.14. Schema bloc a unei unități aritmetico-logice pe 4 biți

Operațiile ALU sunt controlate prin intermediul intrărilor de *selecție-funcție* (M, S_1 și S_0). Intrarea de selecție M va determina modul de operare – aritmetic (M=1) sau logic (M=0) iar S_1 și S_0 va stabili operația propriu-zisă (adunare / scădere / OR / AND …). În general o unitate aritmetico-logică trebuie să realizeze următoarele operații (vezi tabelul 3.2).

M = 0 Logic				
S1	S0	C0	Funcția	Operația (pe bit)
0	0	X	A_iB_i	AND
0	1	X	$A_i + B_i$	OR
1	0	X	$A_i \oplus B_i$	XOR
1	1	X	$(A_i \oplus B_i)'$	XNOR

M = 1 Aritmetic				
S1	S0	C0	Funcţia	Operaţia
0	0	0	A	Transfer A
0	0	1	A + 1	Incrementează A cu 1
0	1	0	A + B	Adună A cu B
0	1	1	A + B + 1	Incrementează suma dintre A şi B cu 1
1	0	0	A + B'	A + complementul faţă de 1 al lui B
1	0	1	A - B	Scade B din A (i.e. B' + A + 1)
1	1	0	A' + B	B + complementul faţă de 1 al lui A
1	1	1	B - A	Scade A din B (sau A' + B + 1)

Tabelul 3.2. Operaţiile executate de ALU

Primul pas în proiectarea unei unităţi ALU pe 4 biţi îl reprezintă proiectarea unui ALU pe un bit (pentru simplitate va fi notat ALU-1). O primă metodă constă în realizarea tabelei de adevăr pentru ALU-1. Întrucât există 6 intrări (M, S_1, S_0, C_0, A_i şi B_i) şi două ieşiri (F_i şi C_{i+1}) aplicarea metodei (*cu creionul pe hârtie*) este dificilă din punct de vedere al timpului consumat. O soluţie alternativă constă în separarea în două module (unitate aritmetică) şi (unitate logică) şi proiectarea individuală a fiecărui modul. Schema bloc a ALU-1 conţine pe lângă cele două module şi un multiplexor 2:1 (vezi figura 3.15).

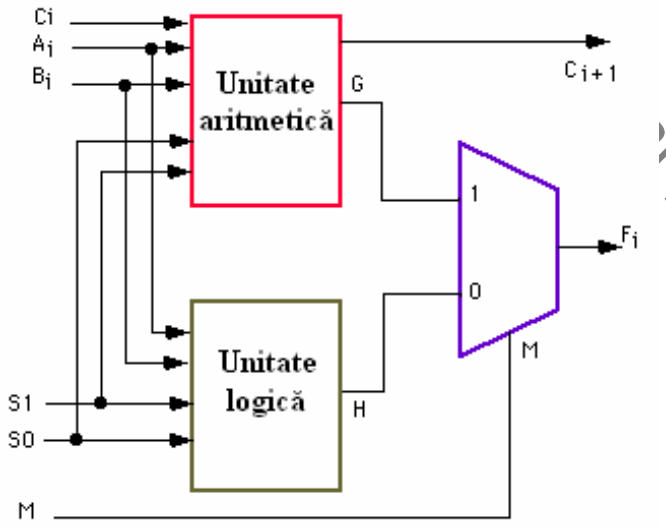


Figura 3.15. Schema bloc a unităţii aritmetico-logice pe 1 bit (ALU-1)

Unitatea logică poate fi realizată folosind un decodificator 2:4 şi porți logice AND, OR, XOR şi NOT (vezi figura 3.16).

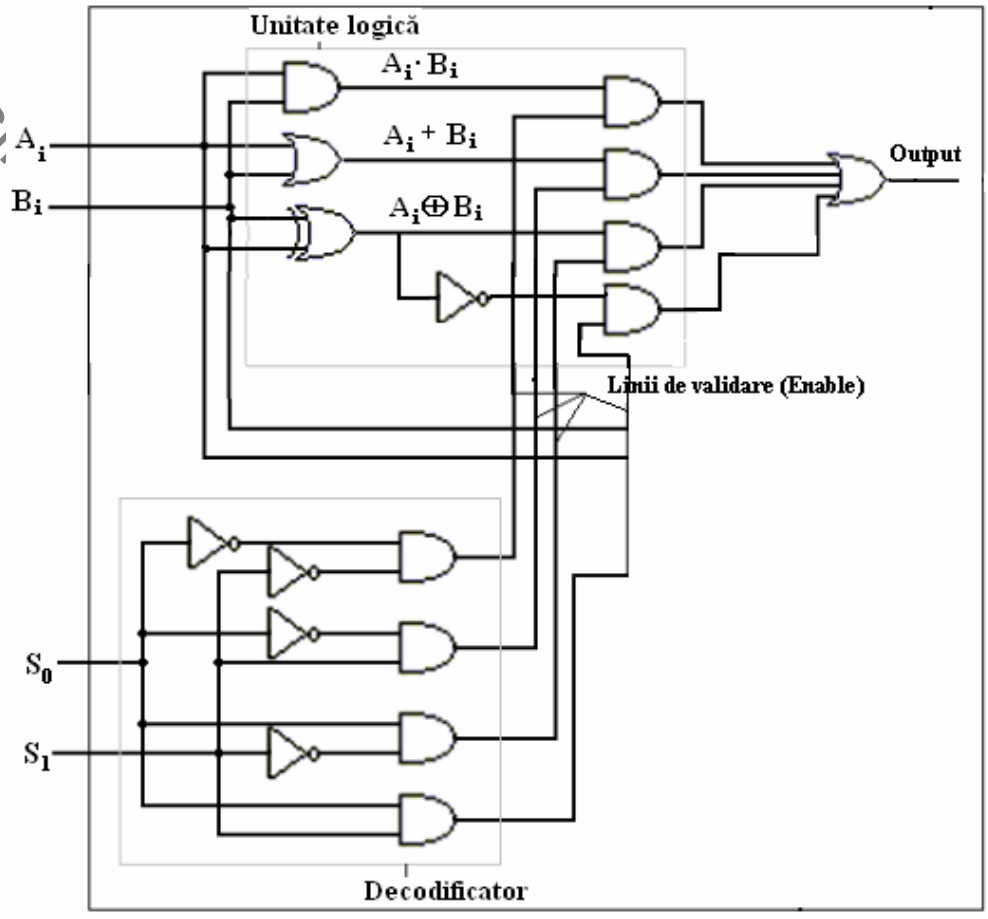


Figura 3.16. Unitatea logică pe 1 bit

Proiectarea unității aritmetice din figura 3.15 se bazează în principiu pe sumatorul complet pe 1 bit (vezi figura 3.10). Unitatea aritmetică trebuie să realizeze în principiu un şir de operații simple: adunare între doi operanzi, incrementarea unui operand, adunarea a doi operanzi şi a unui 1, adunarea la un număr complementul față de 1 a altuia, diferența dintre două numere (vezi tabelul 3.2). Schema bloc a unității de adunare din cadrul ALU-1 este prezentată în figura 3.17. Practic trebuie implementată doar logica de selecție (blocurile A – logic şi B – logic).

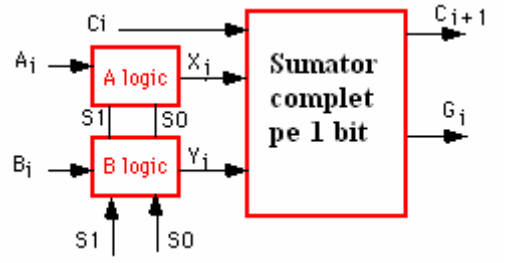


Figura 3.17. Unitatea aritmetică pe 1 bit

În tabelul 3.2 intrarea C_0 (C_i – în figura 3.17) este doar în aparență de selecție. În realitate, în funcție de valoarea lui se va adăuga sau nu 1. Fiecare semnal C_{i+1} va avea rol de transport de intrare pentru A_{i+1} şi B_{i+1} (rangul următor în procesul de adunare).

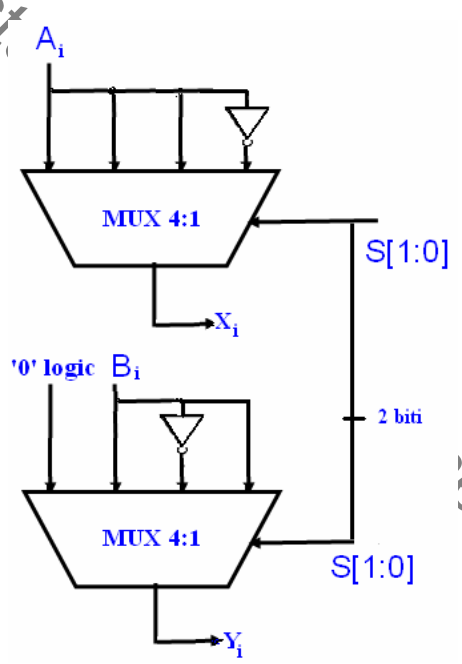


Figura 3.18. Logica de selecție din cadrul unității aritmetice

Operația de deplasare (shift)

Reprezintă operația de translatare cu una sau mai multe poziții binare la stânga sau dreapta. De exemplu, numărul fără semn, reprezentat pe 8 biți, 10010111, prin deplasare la dreapta devine 01001011 (se taie un bit din dreapta şi se adaugă un zero pe prima poziție). Dacă se ține cont de formula

generală de reprezentare a unui număr întreg oarecare pe n+1 digiți în baza b, $N_b = x_n...x_2x_1x_0$, cu $x_n \neq 0$ a cărui valoare numerică este:

$N_b = x_n*b^n+...x_2*b^2+x_1*b^1+x_0$ atunci este foarte simplu de observat că o deplasare cu o poziție la stânga sau dreapta este echivalentă cu o înmulțire respectiv o împărțire cu baza (în cazul acesta b=2). Astfel, valoarea binară 10010111 reprezintă în zecimal numărul 151 iar 01001011 este 75 (partea întreagă a împărțirii 151 / 2).

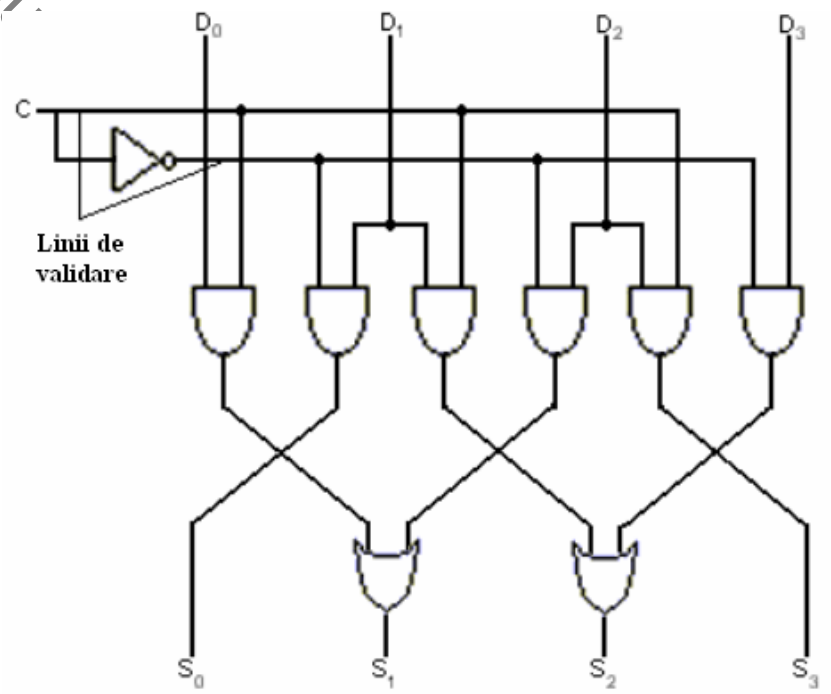


Figura 3.19. Circuit de deplasare (la stânga şi dreapta)

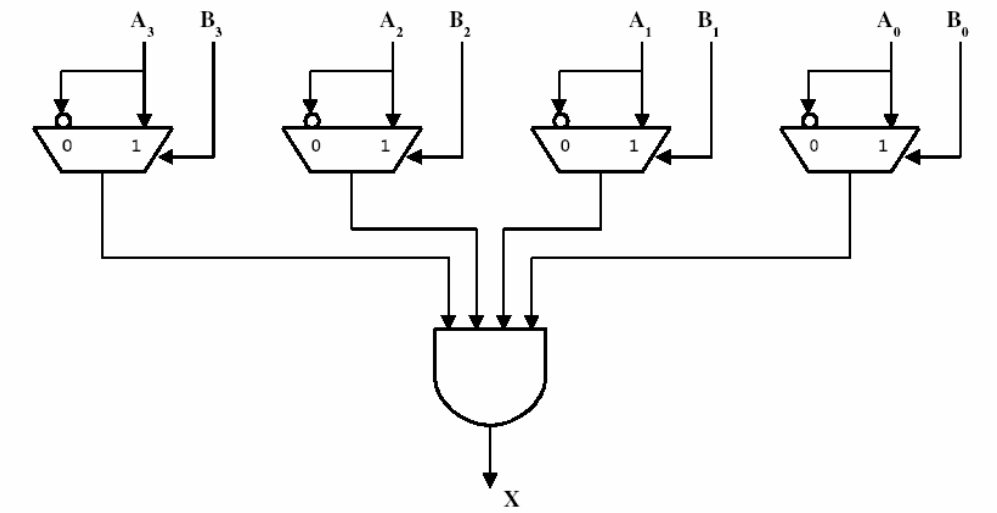
În figura 3.19 este prezentat un circuit de deplasare pe 4 biți. Semnalul C va activa la un moment dat fie deplasarea la dreapta (C=1) fie la stânga (C=0). Dacă deplasarea este la stânga ieşirea circuitului ignoră bitul cel mai din stânga (D_0) şi introduce un zero în coada noului număr (S_3). Dacă deplasarea este la dreapta ieşirea circuitului ignoră bitul cel mai din dreapta (D_3) şi introduce un zero pe prima poziție a noului număr (S_0). Porțile logice SAU deşi primesc două semnale de intrare (D_0 sau D_2, respectiv D_1 sau D_3) permit ca la un moment dat doar unul din cele să furnizeze valoarea în S_1 respectiv S_2. Acest lucru se întâmplă deoarece semnalul *C* comandă intrarea D_0 iar **Not** *C* comandă intrarea D_2.

În acest moment, bazându-ne pe circuitele logice combinaționale anterior prezentate se poate trece la implementarea operațiilor aritmetice de înmulțire respectiv împărțire (mai complexe decât cele de adunare şi scădere).

3.4. EXERCIȚII ŞI PROBLEME

1. Probleme de utilizare a numerelor dintr-o anumita bază (2, 8, 10, 16) pentru – adunarea / scăderea / înmulțirea / împărțirea / număr de biți de 1 / maximul dintre 3 numere – acestora. Se vor folosi funcțiile (clasele) utilizate în rezolvarea seminarului aferent capitolului 2.

2. Referitor la următoarea diagramă logică ce semnifică valoarea 1 la ieşirea **X** ? Un răspuns corect nu necesită mai mult de 10 cuvinte.



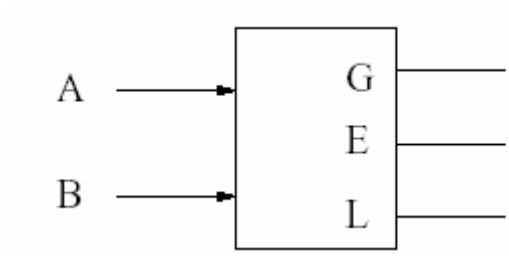
3. Pornind de la cunoştințele de proiectare a unui sumator complet, se doreşte implementarea unui multiplicator (circuit de înmulțire) cu două intrări (fiecare intrare pe 2 biți). Intrările sunt notate `A[1:0]` şi `B[1:0]` iar `Y` fiind rezultatul produs în urma înmulțirii valorilor codificate de A şi B. Notația standard pentru aceasta este: Y = A[1:0]×B[1:0]. Se cere:

a. Care este valoarea maximă care poate fi reprezentată pe 2 biți pentru intrarea `A` (`A[1:0]`) ?
b. Care este valoarea maximă care poate fi reprezentată pe 2 biți pentru intrarea `B` (`B[1:0]`) ?
c. Care este valoarea maximă care poate fi luată de ieşirea `Y` ?
d. Care este numărul minim de biți necesar pentru a putea reprezenta valoarea maximă a ieşirii `Y` ?
e. Realizați tabela de adevăr pentru circuitul de înmulțire descris mai sus. Vor fi patru intrări binare (2 biți pentru intrarea A – `A[1]`, `A[0]`, și alți 2 pentru B – `B[1]` și `B[0]`).
f. Implementați folosind porți logice `AND`, `OR` și `NOT` cel de-al treilea bit al ieşirii Y (`Y[2]`) rezultat din tabela de adevăr de la punctul anterior.

4. Se ştie că poarta logică NAND (ŞI NU) este completă (orice funcție logică poate fi implementată folosind doar porți NAND). Folosind doar porți logice NAND construiți circuite logice combinaționale care să implementeze următoarele funcții. De un real folos ar fi utilizarea regulilor lui DeMorgan.

(a) NOT
(b) AND
(c) OR
(d) NOR

5. Un comparator pe un bit este un circuit (vezi figura de mai jos) având două intrări (fiecare pe 1 bit) A şi B şi trei ieşiri G (mai mare strict), E (Egal) şi L (mai mic strict).



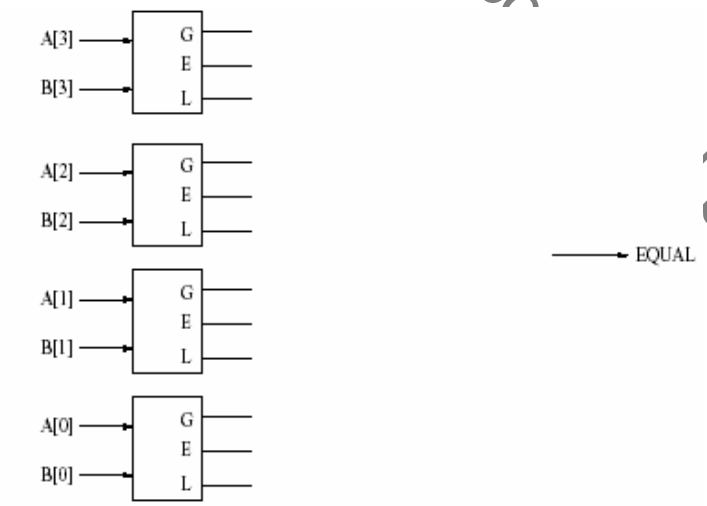
G este 1 dacă A>B, 0 altfel | E este 1 dacă A=B, 0 altfel | L este 1 dacă A<B, 0 altfel

a) Desenați tabela de adevăr pentru comparatorul pe un bit:

A	B	G	E	L
0	0			
0	1			
1	0			
1	1			

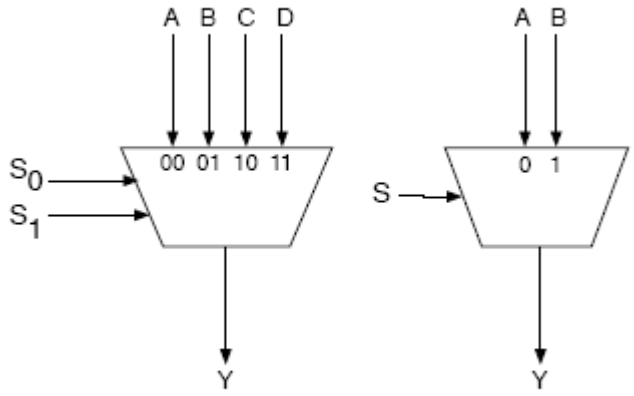
b) Implementaţi G, E şi L folosind porţile logice AND, OR şi NOT.

c) Folosind comparatoare pe 1 bit, construiţi un verificator (comparator pe 4 biţi), care să genereze la ieşire semnalul EQUAL=1 dacă şi numai dacă A[3:0]=B[3:0] cele două intrări A şi B (pe 4 biţi fiecare) coincid.



6.

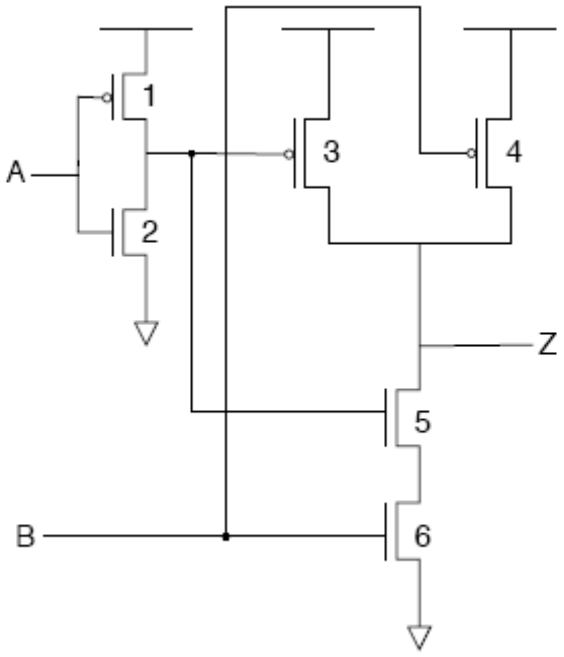
a) Un multiplexor **4:1** (jos, stânga) selectează între 4 intrări folosind două linii de selecție (S_0, S_1). Mai precis, ieşirea multiplexorului va fi egală cu A dacă selecția (S_1S_0) este egală cu 00 (S_1 fiind cel mai semnificativ bit). B se va găsi la ieşire pentru selecția (S_1S_0 = 01), C pentru selecția (S_1S_0 = 10) iar D va fi la ieşirea multiplexorului atunci când ambii biți de selecție vor fi 1. Construiți circuitul de multiplexare 4:1 a cărei funcționare tocmai a fost descrisă folosind multiplexoare **2:1** (jos, dreapta). Intrările în multiplexoarele folosite vor fi etichetate A, B, C şi D, liniile de selecție vor fi notate cu S_0 şi S_1, iar ieşirea circuitului Y.



b) Câte multiplexoare **2:1** sunt necesare pentru construirea unui multiplexor $\mathbf{2^k:1}$ (k>0).

7. În această problemă se cere să se determine pentru toate intrările posibile ale lui A şi B dacă celelalte tranzistoare (etichetate de la 1 la 6) conduc sau nu. Convenim că dacă tranzistoarele (1-6) conduc (circuit închis) atunci au valoarea 1 logic, altfel 0 logic (nu conduc – circuit deschis). De asemenea, indicați valoarea logică a lui Z.

A	*B*	1	2	3	4	5	6	*Z*
0	0							
0	1							
1	0							
1	1							



b) Scrieți expresia logică care descrie ieşirea Z în funcție de intrările A şi B.

8. În această problemă se va construi un circuit combinațional pentru a determina când apare depăşire (*overflow*) la adunarea a două numere în complement față de 2.

a) Se presupune că numerele **a** şi **b**, folosite ca intrări, sunt pe 4 biți: a = $a_3a_2a_1a_0$ şi b = $b_3b_2b_1b_0$.

i. Presupunând primul caz în care a≥0 şi b≥0 se pune întrebarea „*Se obține depăşire la adunarea celor două numere (a şi b) ?*" Scrieți expresia logică corespondentă acestui caz (în termenii biților lui *a* şi

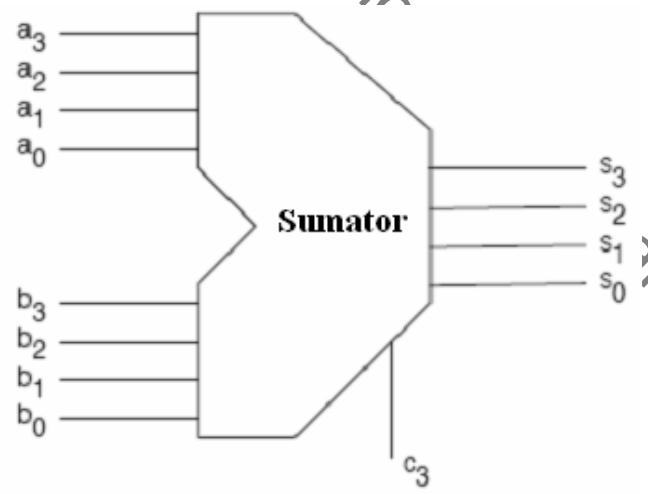
respectiv *b*) care este adevărată doar pentru aceste valori ale lui a şi b (pozitive).

ii. Cerință identică cu cea de la punctul anterior cu deosebirea că $a \geq 0$ şi $b < 0$.

iii. Cerință identică cu cea de la punctul **i.** cu deosebirea că $a < 0$ şi $b \geq 0$.

iv. Presupunând în acest caz că $a < 0$ şi $b < 0$ se pune întrebarea „*Se obține depăşire la adunarea celor două numere (a şi b) ?*" Scrieți expresia logică corespondentă acestui caz (în termenii biților lui *a* şi respectiv *b*) care este adevărată doar pentru aceste valori ale lui a şi b (negative).

b) Se presupune la acest punct că dispunem de un sumator funcțional pe 4 biți având ca intrări pe a ($a = a_3a_2a_1a_0$) şi b ($b = b_3b_2b_1b_0$). Ieşirea sumatorului, tot pe 4 biți este **s**, ($s = s_3s_2s_1s_0$) şi un bit de transport final c_3. Scrieți expresia logică care este adevărată când suma celor două numere a şi b generează depăşire (în funcție de biții lui a, b şi s, şi respectiv bitul de carry, c_3), indiferent de semnul celor două numere. (Indicație: se vor folosi informațiile de la punctul **(a)**). Îmbogățiți circuitul sumator cu porți logice în vederea implementării *semnalului de depăşire* (numit **overflow**).
Acesta va lua valoarea 1 la generarea depăşirii de către sumă.



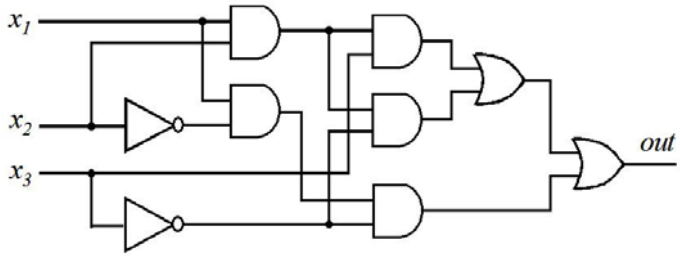
9. a) Câte linii de selecție şi câte linii de ieşire furnizează multiplexoarele următoare ? **Notă:** Numărul liniilor de intrare este specificat în tabelul de mai jos.

Număr linii de intrare	**Număr linii de selecţie**	**Număr linii de ieşire**
32		
16		
5		

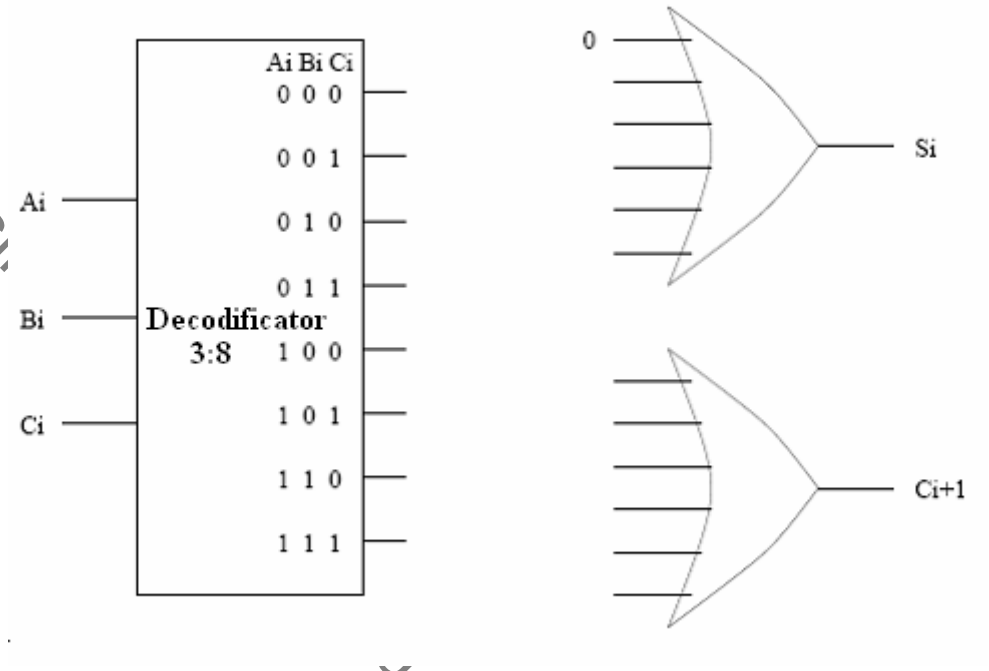
b) Câte linii de ieşire furnizează decodificatoarele următoare ? **Notă:** Numărul liniilor de intrare este specificat în tabelul de mai jos.

Număr linii de intrare	**Număr linii de ieşire**	**Număr linii de ieşire active la un moment dat**
16		
5		
3		

10. Care este expresia logică (booleană) a următorului circuit combinaţional ?



11. Se reaminteşte că un sumator se construieşte din componente individuale (sumator complet pe un bit) care produc un bit de sumă şi unul de transport (*carry*) în funcţie de intrările binare A, B şi bitul *Carry* sosit de la nivelul anterior. Presupunând că dispunem de un **decodificator 3 la 8** şi două **porţi OR cu 6 intrări** (vezi figura de mai jos), se pune întrebarea dacă pot fi conectate astfel încât să obţinem un sumator complet pe un bit. Dacă DA, realizaţi acest lucru. (Indicaţie: dacă o intrare în poarta OR nu este necesară, atunci aceasta poate fi înlocuită cu 0, neafectând rezultatul final).



12. Răspundeți la următoarele întrebări:

a) Care ar fi relația descrisă printr-o expresie logică între tranzistoarele de tip P şi N ?

b) Prin ce se caracterizează circuitele logice combinaționale ? Dar cele secvențiale ?

c) Cum funcționează (descrieți pe scurt) decodificatoarele, sumatoarele şi multiplexoarele ?

4. ELEMENTE PRIMARE DE MEMORARE. REGIŞTRII. MEMORII. AUTOMATE CU NUMĂR FINIT DE STĂRI

4.1. STRUCTURI LOGICE SECVENȚIALE

Se defineşte circuit de memorie orice componentă semiconductoare care implementează funcția de memorare. Un circuit secvențial este un CLC care are memorie şi îi permite să memoreze „istoria" lui (vezi figura 4.1). Ieşirea este o funcție de intrări şi starea curentă. Funcția este calculată de către circuitul logic combinațional. Starea este reținută în *elementul de memorare*. Starea următoare se determină, de asemenea, în funcție de intrări şi starea curentă.

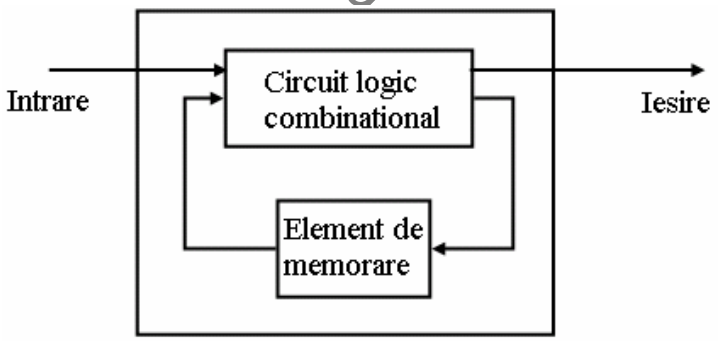


Figura 4.1. Circuit logic secvențial

Sunt de 2 feluri de circuite logice secvențiale: **sincrone** (CLSS) şi **asincrone** (CLSA).

- CLSS – sunt caracterizate de faptul că există o trecere discretă a timpului. Transferul de informație se produce numai pe durata tactului.
- În cazul CLSA nu există impulsuri de tact şi momentele succesive de timp sunt date de caracteristicile circuitului. Memoria joacă rolul unei linii de întârziere.

4.1.1. BISTABILUL R-S

Reprezintă un element simplu de memorare care poate reține un bit de informație, circuitele bistabile putând avea la ieşire două stări stabile (0 logic şi 1 logic) un timp nedefinit şi doar o comandă externă putând declanşa modificarea stării logice a ieşirii. Există mai multe variante de implementare, în figura 4.2 fiind prezentată una dintre acestea. Este realizat din două porți de tip *NAND*. Ieşirea fiecărei porți se conectează la intrările celeilalte. În mod normal (cel mai mult timp) celelalte două intrări (*S* şi *R*) sunt în starea 1 logic. După cum se va vedea în continuare, există pentru intervale scurte de timp, situații când fie S fie R pot fi în 0 logic, dar nu simultan.

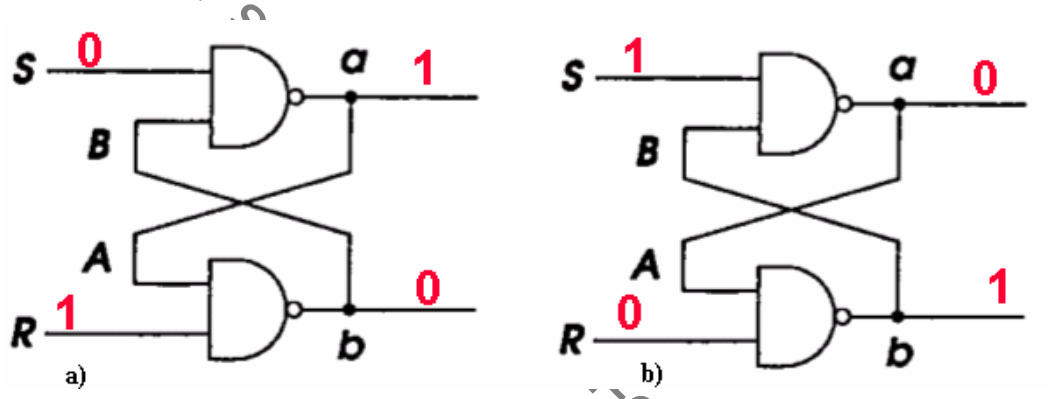


Figura 4.2. Bistabil R-S: reține două stări stabile a) (a=1) şi b) (a=0)

În cazul a) din figura 4.2, ieşirea **a** poate fi setată pe 1 prin setarea temporară a lui S pe 0 şi menținerea (îndelungată) a intrării R pe 1. Odată ce ieşirea **a** este 1, practic intrarea A în poarta NAND din josul figurii este 1, R de asemenea este 1, rezultând la ieşirea **b** valoarea 0, care înseamnă de fapt 0 la intrarea B în poarta de sus. Astfel, la ieşirea **a** se va găsi în continuare 1 necondiționat de valoarea lui S. După revenirea lui S în 1 la ieşirea **a** rămâne tot 1. Se spune astfel că bistabilul R-S stochează valoarea 1.

Invers, în cazul b) ieşirea **a** poate fi setată pe 0 prin păstrarea intrării S la valoarea 1 şi setarea temporară a intrării R pe 0. Analog cu cazul a), la ieşirea b şi implicit la intrarea B se va afla valoarea 1 logic. Ținând cont de faptul că S este permanent în 1 atunci la ieşirea **a** se va afla 0, care propagată pe intrarea A a porții de jos va face inutilă prezența lui R în 0, rezultatul în **b** fiind 1. Astfel, după revenirea lui R în 1 la ieşirea **a** a bistabilului va fi tot 0. Se spune că bistabilul R-S stochează valoarea 0.

În tabelul 4.1 este sintetizată funcţionarea bistabilului R-S în funcţie de configuraţia stării curente (notată **Q**) şi cea viitoare (notată $\mathbf{Q^+}$).

Q	Q^+	S	R
0	0	1	*
0	1	0	1
1	0	1	0
1	1	*	1

Tabelul 4.1. Configuraţia intrărilor R şi S pentru tranziţia stărilor din Q în Q^+

În cazul tabelului anterior trebuie făcute două precizări:

- Prima se referă la faptul că starea Q reprezintă valoarea ieşirii **a** a bistabilului R-S la un moment t_0 iar starea Q^+ constituie valoarea ieşirii **a** la momentul $t_0 + \Delta t$.
- A doua se referă la valoarea * care trebuie înţeleasă astfel: iniţial valoarea lui R sau S (în funcţie de coloana pe care apare) este 0, urmând ca ulterior la momente de $t_0 + 2 \cdot \Delta t$, R sau S să revină în 1.

Bistabilul R-S are *autonomie limitată*: el depinde de starea lui anterioară şi de intrări. Ieşirile se schimbă numai pentru anumite valori ale intrărilor. Realizează funcţia de memorare pentru că variaţia unor intrări în anumite cazuri nu are efect nici asupra ieşirilor şi nici asupra stărilor. Există două tipuri de bistabili R-S: varianta asincronă (vezi figura 4.2) şi varianta sincronă (vezi figura 4.3).

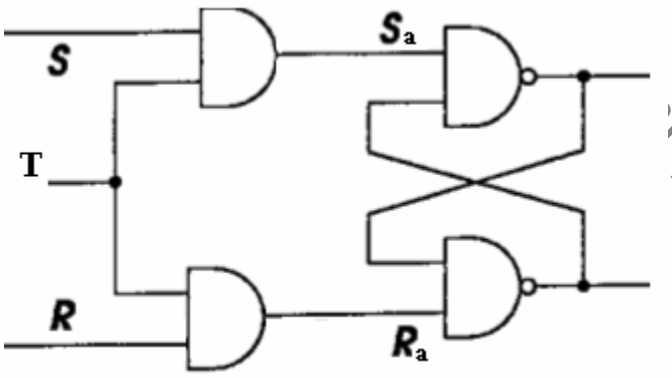


Figura 4.3. Bistabil R-S sincron (intrările R_a şi S_a sunt intrările aferente bistabilului R-S asincron, iar T este semnalul de ceas)

Bistabilul R-S prezintă şi unele dezavantaje:

- Apar stări interzise (R=0 şi S=0)

- În cazul bistabilului R-S sincron apar limitări datorate impulsului de tact.
- Are două intrări de comandă pentru a comanda o singură ieşire (care este tot una cu starea)

4.1.2. BISTABILUL DE TIP D (DELAY).

Reprezintă o extensie a bistabilului de tip R-S şi este un bistabil cu rol de întârziere. Există două intrări: una de date – **D** şi una de validare a procesului de memorare – **WE** (write enable). Dacă WE este egal cu 1 atunci ieşirea bistabilului (fie aceasta Q) este setată la valoarea D (practic semnalul de ieşire este identic cu cel de intrare cu mențiunea că este întârziat cu un interval de timp egal cu timpul de propagare prin două porți logice fundamentale ŞI-NU – vezi figura 4.5). Ieşirea rămâne la valoarea setată până la o nouă setare a semnalului WE pe 1 (când la ieşire va fi noua valoare a lui D).

Bistabilul de tip D (vezi figura 4.4) nu are variantă de funcționare *fără modificare de stare* ci, la fiecare impuls de tact primit (activare a semnalului WE[3]), valoarea preluată de bistabil pe intrarea D este cea care determină următoarea stare la ieşire. Pentru a păstra valoarea la ieşire nemodificată, cât timp nu se primeşte impuls de tact, este necesar să menținem intrarea D egală cu valoarea curentă de pe ieşire. Această funcționare este asigurată de conexiunile de reacție, care leagă fiecare ieşire Q la intrarea D a aceluiaşi bistabil.

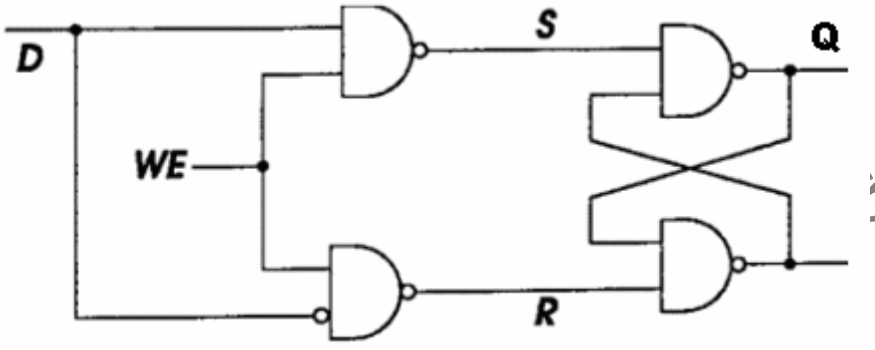


Figura 4.4. Bistabil de tip D

[3] Semnalul WE devine 1 logic.

Ecuaţia logică de stare a bistabilului de tip D este: $\mathbf{Q(t_{n+1}) = D(t_n)}$, unde $\mathbf{t_{n+1} - t_n}$ este maxim o perioadă de tact procesor (T_{CLK}). Ieşirea Q reprezintă intrarea D întârziată cu maxim un tact.

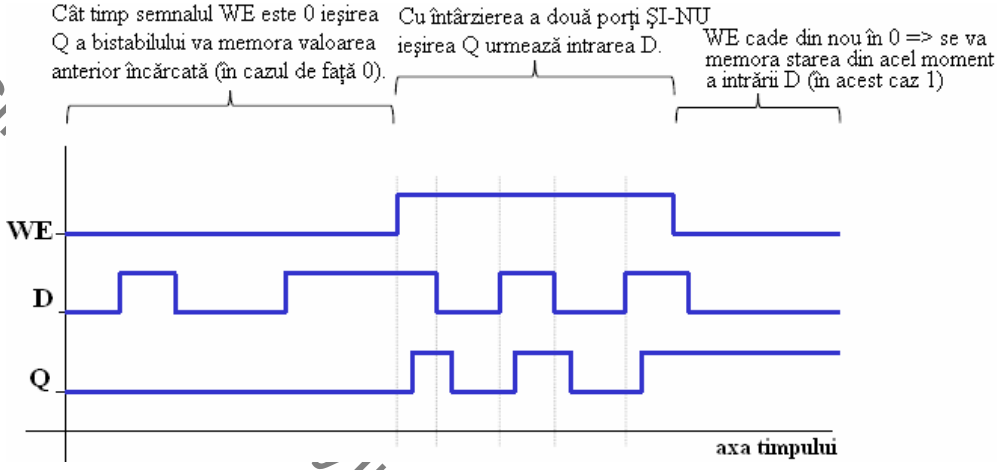


Figura 4.5. Bistabil de tip D – cronogramă

4.1.3. REGIŞTRII.

Un **registru** este un ansamblu de bistabili şi, eventual, porţi logice care realizează schimbările de stare ale bistabililor. Grupul de bistabili care formează registrul pot fi încărcaţi simultan, sub acţiunea unui impuls de tact unic. Un **registru de *n* biţi** are *n* bistabili şi poate memora şi / sau deplasa orice informaţie reprezentabilă pe *n* biţi. Porţile logice au rolul de a controla când şi cum o informaţie nouă este transferată în registru.

Componentele unui sistem digital sunt complet definite de regiştrii pe care îi conţin şi de operaţiile care se execută asupra datelor lor. Operaţiile executate asupra datelor memorate în regiştrii se numesc **microoperaţii**. Exemple: deplasare, numărare, încărcare, ştergere. Pentru executarea fiecăreia dintre aceste microoperaţii există structuri bine determinate ale regiştrilor respectivi. În continuare se vor descrie structurile unora dintre aceşti regiştri.

Încărcarea regiştrilor

Funcţia unui registru cu încărcare paralelă este de a prelua valorile logice puse pe intrările de date, simultan pentru toate poziţiile binare la

primirea aceluiaşi impuls de ceas, şi de a stoca temporar configuraţia binară respectivă în scopul unui acces uşor la ea în vederea prelucrării. Registrul paralel (de stocare / tampon) [Fil96] – vezi figura 4.6, este format din *n* bistabili de tip D acţionaţi sincron de un tact comun.

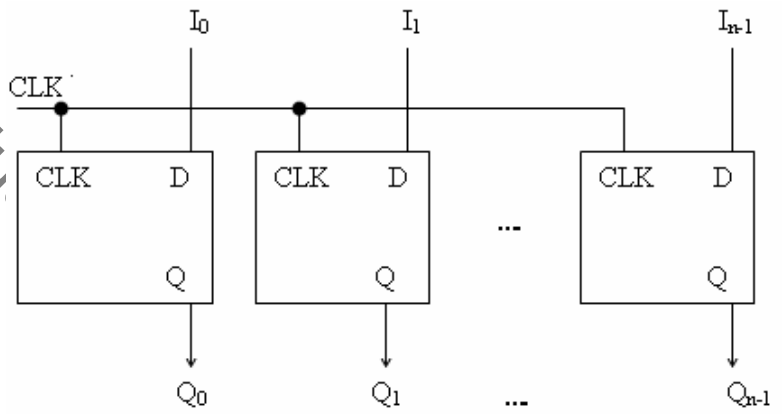


Figura 4.6. Schema generală a unui registru paralel

În momentul aplicării tactului, cuvântul binar de *n* biţi prezent la intrările I_0, I_1, ..., I_{n-1} este înscris în cele *n* celule de memorie şi poate fi citit la ieşirile Q_0, Q_1, ..., Q_{n-1}.

Registrul de deplasare

Registrul de deplasare serie este format din *n* bistabili de tip D conectaţi în cascadă (ieşirea dintr-un bistabil este intrare în următorul) [Fil96]. În plus, funcţionarea este sincronizată prin faptul că toate intrările de ceas sunt legate la aceeaşi sursă. Astfel, toţi bistabilii vor primi la acelaşi moment impulsul de tact care determină deplasarea informaţiei. Un registru capabil să deplaseze, adică să translateze cu o poziţie informaţia memorată de bistabilii săi se numeşte **registru de deplasare**. În funcţie de construcţia registrului, operaţia de mutare (*shift*) se poate face spre stânga, spre dreapta sau în ambele direcţii.

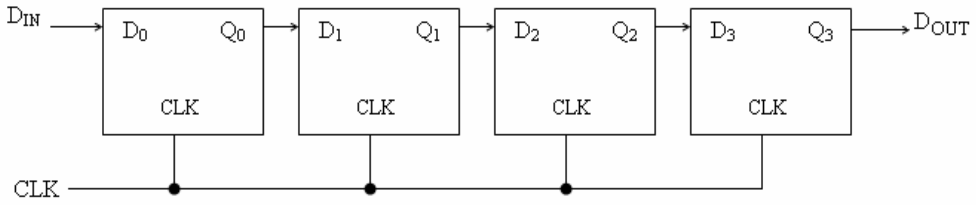


Figura 4.7. Schema generală a unui registru serie

Intrarea D_{IN} conţine informaţia care se va memora în primul bistabil, iar ieşirea D_{OUT} va conţine vechiul conţinut al ultimului bistabil al registrului de deplasare analizat. Se observă că informaţia D_{IN} ajunge la ieşirea D_{OUT} după *n* impulsuri de tact.

Regiştrii combinaţi (cu funcţii multiple)

Cele două tipuri de registre tratate anterior sunt utilizate în aplicaţii în care transferul datelor se face fie numai paralel fie numai serie. Registrele combinate permit trecerea de la transferul paralel la cel serie şi invers. În figura 4.8 se prezintă un registru combinat pe 4 biţi [Fil96].

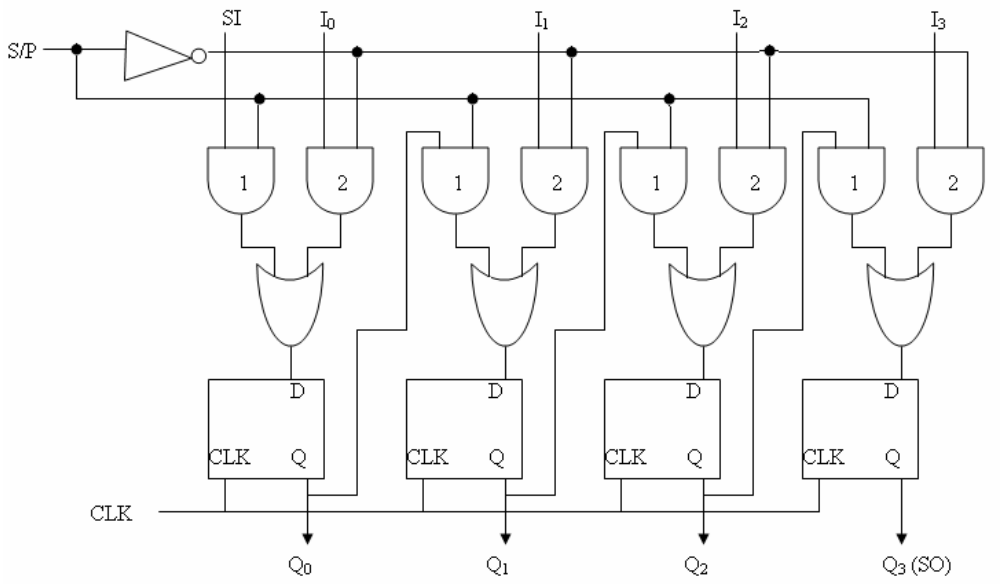


Figura 4.8. Schema generală a unui registru paralel-serie sau serie-paralel

- ✓ Dacă semnalul S/P (comportament de registru serie / paralel) este 0, sunt deschise porţile 2 şi datele de intrare I_0, I_1, I_2 şi I_3 au acces la intrările celor 4 bistabile. Încărcarea paralel are loc în momentul aplicării impulsului de ceas (CLK).
- ✓ Dacă S/P=1, registrul realizează o deplasare a datelor de la stânga la dreapta, cu câte un bit pentru fiecare impuls de ceas.

Registrul funcţionează atât ca şi **convertor paralel-serie**, datele fiind introduse paralel la intrările I_0, I_1, I_2 şi I_3, fiind extrase serie la ieşirea SO (*serial output*), cât şi ca un **convertor serie-paralel**, datele se introduc de o manieră serială la intrarea SI (*serial input*) şi sunt extrase paralel la ieşirile Q_0, Q_1, Q_2 şi Q_3.

Referitor la regiştrii utilizaţi de către calculatorul LC-3 se disting:

- ✓ 8 regiştri de uz general $R_0 \div R_7$, IR (instruction register), PC (program counter), MAR (registru de adresă al memoriei) – toate pe 16 biţi şi,
- ✓ regiştri booleeni pe 1 bit: N – semn negativ, P – semn pozitiv, Z – pentru identificarea numerelor nule.

Dintre regiştri pe 16 biţi, registrul PC trebuie să îndeplinească funcţiile de încărcare paralelă şi cea de numărător, regiştrii de uz general trebuie să fie capabili de deplasări, încărcare paralelă, iar pentru regiştrii IR, MAR şi MDR este necesară doar încărcarea paralelă.

Trebuie menţionat că se poate opera nu numai asupra regiştrilor (ca şi entităţi atomice) ci şi asupra câmpurilor din interiorul unui registru (de exemplu: IR[3:0]).

4.1.4. MEMORIA

Reprezintă o zonă de date de capacitate mai mare decât regiştrii procesorului dar mai lentă decât aceştia. Se cunosc următoarele tipuri de memorii:

- **ROM** (*read only memory*) – memorii fixe, nevolatile, în care informaţia se păstrează şi după întreruperea alimentării. Circuitele ROM sunt circuite pur combinaţionale, celula de memorie fiind un tranzistor programat sau nu, amplasat la intersecţia unei linii cu a unei coloane, din matricea de memorie (vezi figura 4.10). În ele se înscrie softul de bază al unui sistem de calcul: *sistemul de operare*, BIOS (rutina de iniţializare a sistemului de calcul imediat după alimentarea cu energie – teste de memorie, verificare periferice). Memoriile ROM pot fi doar citite nu scrise (scrierea se face o singură dată în momentul fabricării).
- **PROM (*programmable ROM*) –** poate fi programat de producător sau utilizator la prima folosire a dispozitivului.
- **UVEPROM (Ultraviolet Erasable PROM) –** sunt identice cu memoriile PROM cu deosebirea că oferă posibilitatea de ştergere şi reprogramare prin expunerea la raze ultraviolete. Necesită un echipament special pentru programare.
- **EEPROM (Electrical Erasable PROM) –** sunt identice cu memoriile UVEPROM cu deosebirea că ştergerea se face prin aplicarea unor semnale electrice speciale de tensiune ridicată (30V).
- **FLASH ROM –** este o memorie specială de tip EEPROM care poate fi ştearsă sau programată în timpul funcţionării în circuit (aplicaţii cu

microcontrollere în care se încarcă un cod obiect gata de execuţie). Odată programat conţinutul rămâne nemodificat chiar şi după eventuale anomalii datorate căderii tensiunii de alimentare.

- **RAM (***random access memory***) –** este o memorie care poate fi scrisă sau citită (atâta timp cât este alimentată cu energie). Informaţia se pierde după oprirea alimentării. Accesul la memorie se permite numai în anumite momente de timp validate de un semnal de tip acces la memorie (*memory request*). Există două tipuri majore de memorii RAM:
  - **SRAM (Static RAM) –** mai rapide, mai fiabile dar mai scumpe per unitate de octet memorat. Uzual sunt folosite la implementarea memoriilor *cache*. De asemenea, consumă mai multă energie.
  - **DRAM (Dynamic RAM) –** lentă şi necesită regenerare (celula de memorie o reprezintă un condensator care se descarcă în timp); ieftină per unitate de octet memorat. Memoria principală este de tip DRAM. Din punct de vedere al evoluţiei în timp a memoriilor DRAM se cunosc următoarele etape:
    - EDO (*Extended Data Out RAM*) – cu 10% până la 20% mai rapidă decât primele memorii DRAM.
    - SDRAM (*Synchronous DRAM*) – mai rapide cu aproape 25% decât memoriile EDO RAM.
    - DDR sau SDRAM II (*Double Data Rate SDRAM*) – de două ori mai rapide decât memoriile SDRAM.
    - RDRAM (*Rambus DRAM*) – dezvoltate de către firma Rambus Inc., sunt de aproape zece ori mai rapide decât memoriile DRAM.
    - SLDRAM (*Synclink DRAM*) – este principalul competitor din punct de vedere tehnologic al memoriilor RDRAM.

Creşterea decalajului dintre viteza procesoarelor şi timpul de acces la memorie a impus introducerea unui sistem ierarhic de memorie (vezi figura 4.9), pentru a nu face simţită la nivelul performanţei globale a sistemului încetineala cu care se accesează memoria. Practic, cu cât capacitatea memoriei creşte, cu atât scade timpul de acces la memorie şi se ieftineşte preţul per unitate de octet memorat. Memoria **cache** este o memorie situată din punct de vedere logic între CPU (unitatea centrală de procesare) şi memoria principală, mai mică, mai rapidă şi mai scumpă (per byte) decât aceasta şi gestionată – în general prin hardware – astfel încât să existe o cât mai mare probabilitate statistică de găsire a datei accesate de către CPU, în cache. Aşadar, cache-ul este adresat de către CPU în paralel cu memoria principală (MP): dacă data dorită a fi accesată se găseşte în cache, accesul la MP se abortează, dacă nu, se accesează MP cu penalizările de timp impuse

de latența mai mare a acesteia, relativ ridicată în comparație cu frecvența de tact a CPU. Oricum, data accesată din MP se va introduce şi în cache.

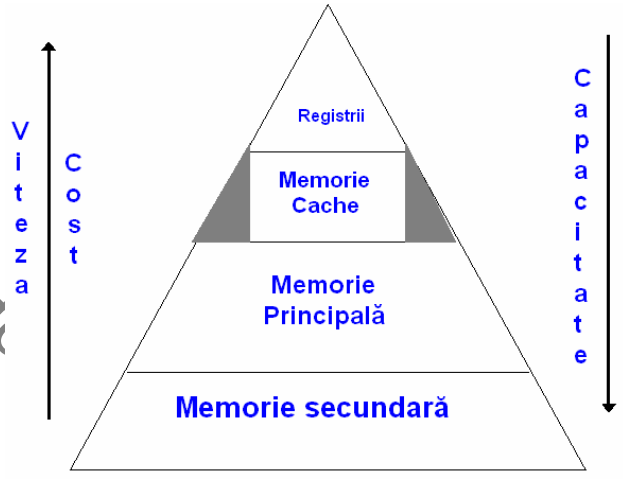


Figura 4.9. Ierarhizarea memoriei într-un sistem de calcul

Memoria este caracterizată de un număr mare de locații de dimensiune fixă fiecare. Cu n biți de adresă pot fi adresate 2^n locații de memorie. Astfel, cu 24 de biți de adresă pot fi accesate 2^{24} locații, adică 16 Megalocații – 16777216 locații. Dacă fiecare locație reține 1 octet atunci spațiul de memorie acoperit este de 16 MBytes iar dacă locația conține un cuvânt de 4 octeți rezultă că spațiul de memorie adresabil este de 16·4=64 MBytes.

Calculatoarele sunt adresabile (pot extrage cuvinte de dimensiuni variabile) în mod normal pe cuvânt sau pe octet. Există însă procesoare cu instrucțiuni care solicită încărcarea / scrierea din / în memorie a unui semicuvânt (2 octeți) sau a unui dublu cuvânt (8 octeți) (vezi **lh** – *load high* sau **ld** – *load double* la procesorul MIPS). În general un cuvânt este scris sau citit la un moment dat. În cazul în care se doreşte accesul în scriere sau citire la un singur octet trebuie cunoscută convenția de reprezentare de pe mașina respectivă (*big* sau *little endian*).

În figura 4.10 este descris modul de implementare a unei memorii cu 4 locații, fiecare conținând un bit. Bitul de informație este reținut în bistabili de tip D. De regulă, fiecare locație conține w biți (în figura 4.10 – w=1). Dacă w=8 atunci memoria este adresabilă pe octet. Pentru adresarea celor n locații de memorie sunt necesari $\lceil \log_2 n \rceil$ biți de adresă dacă se consideră n o putere a lui 2, altfel $\lceil \log_2 n \rceil + 1$. Circuitul decodificator va selecta doar o locație de memorie din cele n, la un moment dat.

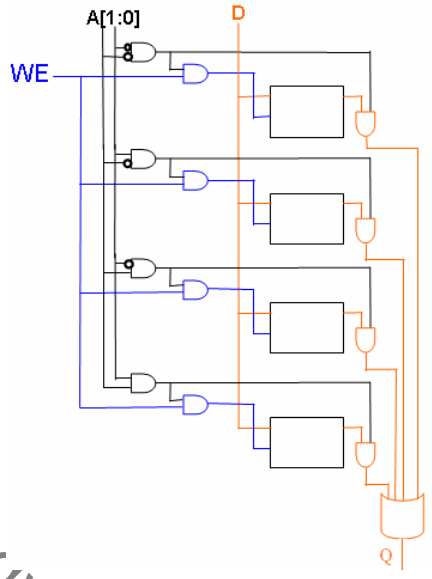


Figura 4.10. Schema generală a unei memorii pe un bit (soluția nescalabilă)

În realitate semnalul de ieșire nu poate fi obținut dintr-o poartă SAU cu 2^n intrări. Soluția de scalabilitate a ieşirii compusă din cele 2^n intrări presupune folosirea de porți SAU cu 2 intrări ca în figura 4.11. Se impune astfel observația conform căreia *timpul de citire al locației 0 este mai lung decât timpul de citire al ultimei locații* (2^n*-1*) – informația de la locația 0 are de parcurs mult mai multe porți SAU decât oricare alta.

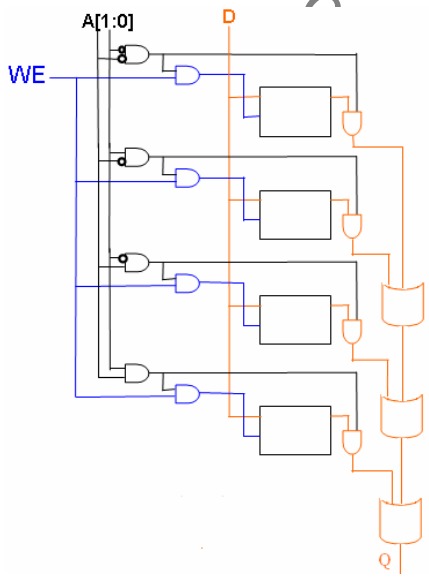


Figura 4.11. Schema generală a unei memorii pe un bit (soluția scalabilă)

În figura 4.12 este prezentată o memorie tot cu 4 locații, fiecare locație având de data aceasta un număr de 3 biți ($D_2 \div D_0$). Numărul de biți de adresă este $[\log_2 4]$, adică 2 – ($A_1 \div A_0$). Validarea preluării datelor – scrierii sau citirii se face cu semnalul de validare WE. Se remarcă aceeaşi observație de nescalabilitate a porților SAU aflate la ieşirile de date $D_{<2>}$, $D_{<1>}$, $D_{<0>}$.

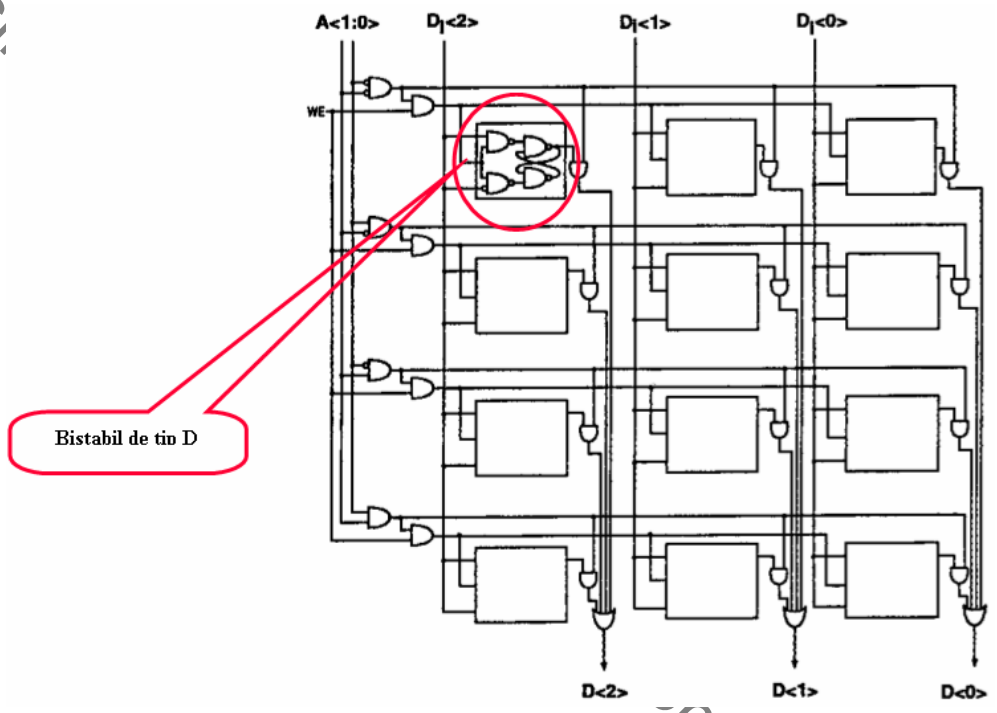


Figura 4.12. Schema generală a unei memorii pe 3 biți

În figura 4.13 este ilustrat modul de implementare a unei memorii de 8Kocteți folosind circuite de 2K intrări adresabile 4 biți. Semnalul CS (chip select) dacă este setat pe 1 este validată adresarea, citirea şi scrierea unui cip. Este necesară gruparea pe fiecare linie a două circuite pentru a realiza o memorie adresabilă pe octet (crearea unui banc de 2Kocteți). Circuitele primare având 2K intrări rezultă necesitatea a 11 biți de adresă ($A_{10} \div A_0$), ceilalți doi biți ($A_{12} \div A_{11}$) din totalul de 13 biți necesari pentru adresarea unui spațiu de 8K intrări sunt folosiți pentru selecția bancului de două circuite primare.

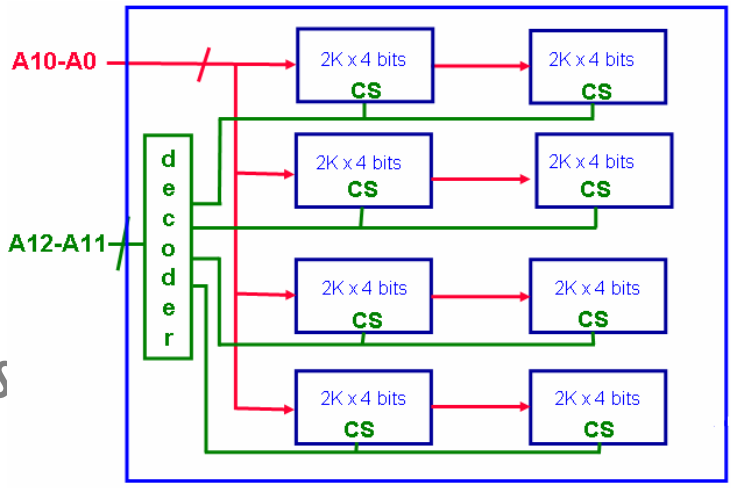


Figura 4.13. Implementarea unei memorii de 8Kocteți folosind circuite de 2K intrări adresabile 4 biți

Aplicație 1: Descrieți funcționarea circuitului secvențial din figura următoare cunoscând că la momentul inițial ieşirile $Q_3Q_2Q_1Q_0$ sunt 0000.

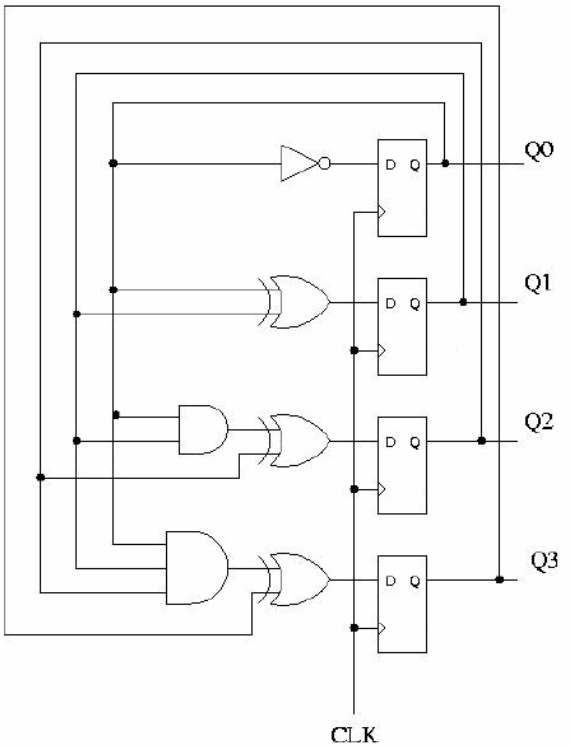


Figura 4.14. Schema unui numărător pe patru biți

Pentru rezolvare se va realiza tabelul de valori al ieşirilor $Q_3Q_2Q_1Q_0$.

Valoarea ieşirilor $Q_3Q_2Q_1Q_0$ la momentul *t*				Valoarea ieşirilor $Q_3Q_2Q_1Q_0$ la momentul următor *t'*			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

Tabelul 4.2. Comportamentul automatului secvențial în funcție de intrări

După cum se poate observa datorită inversorului de la intrarea lui Q_0, acesta va schimba starea cu fiecare impuls de tact. De asemenea, Q_1 va schimba starea abia după două impulsuri de tact (când Q_0 a schimbat din 0 în 1 şi apoi înapoi din 1 în 0). Porțile AND au rolul de a facilita tranzitarea dintr-o stare în alta a lui Q_2 abia după 4 perioade de tact şi respectiv a lui Q_3 abia după 8 perioade de tact. Raționamentul poate continua şi se va ajunge la tabelul de adevăr de mai sus, care de fapt descrie funcționarea unui **circuit numărător saturat** pe 4 biți.

În general, un numărător este un dispozitiv care memorează de câte ori un eveniment particular sau proces a avut loc atunci când s-a activat semnalul de tact. În practică sunt folosite atât numărătoare bazate pe incrementare cât şi numărătoare bazate pe decrementare. Anticipând puțin, unitatea de control a fiecărei arhitecturi de procesare dispune de un registru numit *Program Counter* (numărător de program) care reține adresa următoarei instrucțiuni din program. În funcție de adresabilitatea memoriei şi de numărul de octeți pe care e reprezentată fiecare instrucțiune, acest

numărător va fi incrementat după aducerea din memorie a câte unei instrucţiuni cu o anumită valoare (1, 2, 4, 8), dar de fiecare dată aceeaşi.

4.2. AUTOMATE SECVENŢIALE ŞI PROGRAMABILE

Automatul cu număr finit de stări presupune o desfăşurare automată a unui număr finit de secvenţe. În informatică, automatele finite sunt folosite pe larg în modelarea comportamentului aplicaţiilor, ingineria software, compilatoare, în studiul computaţiei şi limbajelor şi în proiectarea sistemelor digitale hardware (automatele cu număr finit de stări se folosesc în structurile hardware de predicţie aferente ramificaţiilor condiţionate din program, mecanisme de confidenţă în vederea unei evacuări / inserări selective în structuri de predicţie a salturilor indirecte de tip Target Cache, dar şi în reducerea decalajului dintre viteza procesoarelor şi timpul de acces la memorie prin concepte de tip *selective victim cache*).

Un **automat finit** (AF) sau o **maşină cu stări finite** este un model de comportament compus din *stări*, *tranziţii* şi *acţiuni* [Wik]. O stare stochează informaţii despre trecut, adică reflectă schimbările intrării de la iniţializarea sistemului până în momentul de faţă. O tranziţie indică o schimbare de stare şi este descrisă de o condiţie care este nevoie să fie îndeplinită pentru a declanşa tranziţia. O acţiune este o descriere a unei activităţi ce urmează a fi executată la un anumit moment. Există câteva tipuri de acţiuni:

- Acţiune *de intrare* executată la intrarea într-o stare.
- Acţiune *de ieşire* executată la ieşirea dintr-o stare.
- Acţiune *de intrare de date* executată în funcţie de starea prezentă şi de datele de intrare.
- Acţiune *de tranziţie* executată în momentul unei tranziţii.

Logica automatelor finite stabileşte că ieşirea şi starea următoare a unui automat finit este o funcţie de intrare şi de starea curentă. Logica unui AF este prezentată în figura 4.15 [Wik].

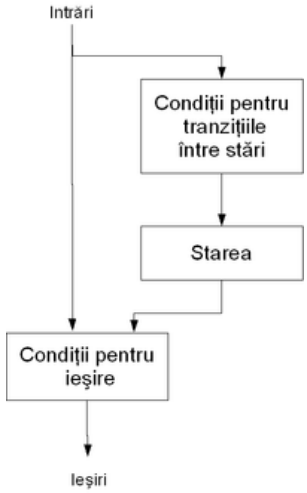


Figura 4.15. Logica automatelor finite

Automatul finit poate fi reprezentat printr-o *diagramă de stări* (sau *diagramă de stări şi tranziţii*) ca în figurile 4.16 şi 4.17 [Wik]. În plus, se folosesc şi tabele de tranziţie. Cea mai comună reprezentare este dată mai jos: prin combinaţia stării curente (B) şi a condiţiei (Y) se determină starea următoare (C). Informaţii complete privind acţiunile pot fi adăugate doar ca note de subsol. În limbaj natural funcţionarea automatului următor s-ar descrie astfel: *Din starea B dacă se îndeplineşte condiţia Y se trece în starea C*.

Starea curentă / Condiţia	Starea A	**Starea B**	Starea C
Condiţia X	...	...	...
Condiţia Y	...	**Starea C**	...
Condiţia Z	...	...	...

Tabelul 4.3. Tabel de tranziţie într-un automat cu număr finit de stări – caz general

Figura 4.16 descrie funcţionarea unui automat simplu de închidere / deschidere a unei uşi (de supermarket, de autobuz, etc.). Practic automatul se poate afla în două stări posibile (DESCHIS sau ÎNCHIS). În fiecare din cele două stări se ajunge datorită unei *Condiţii de tranziţie* (declanşată practic de „*scurgerea*” unui interval de timp sau de apăsarea unui buton, sau de un senzor de mişcare, etc.) iar la intrarea în fiecare stare are loc o *acţiune de intrare*.

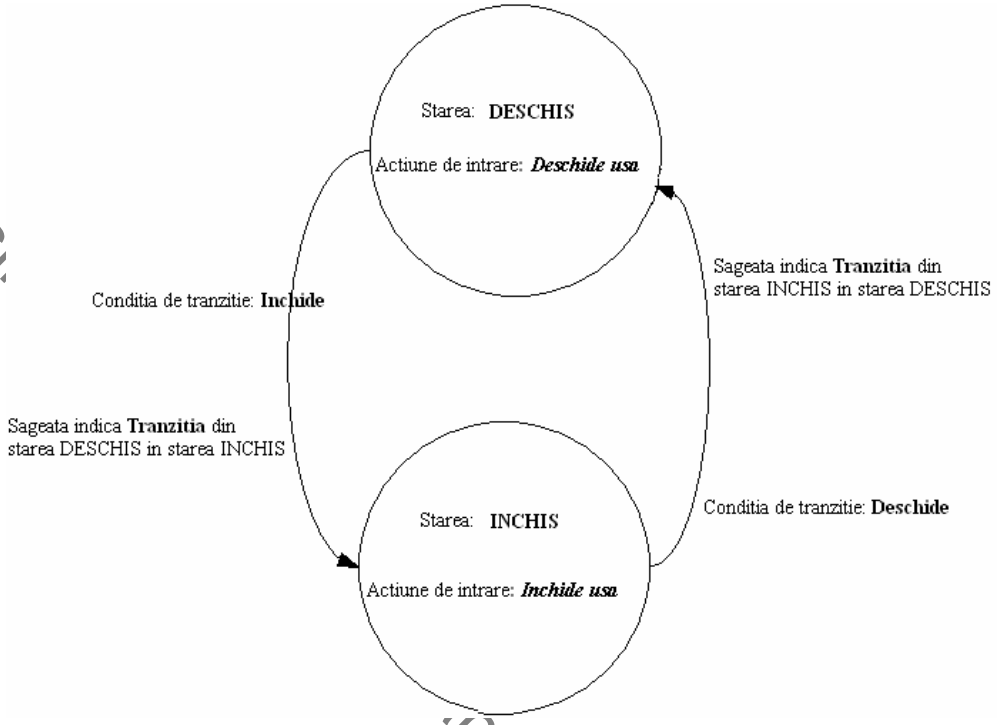


Figura 4.16. Diagrama de stări şi tranziţii – automat simplu de închidere / deschidere a unei uşi

Pentru o şi mai bună exemplificare, şi pentru crearea unei legături cu cursul care urmează acestuia introductiv, şi anume cel de *Organizarea şi proiectarea microarhitecturilor*, se prezintă funcţionarea unui automat finit pe doi biţi (de tip numărător saturat, folosit de cele mai multe structuri de predicţie implementate în cadrul procesoarelor actuale) [Flo05].

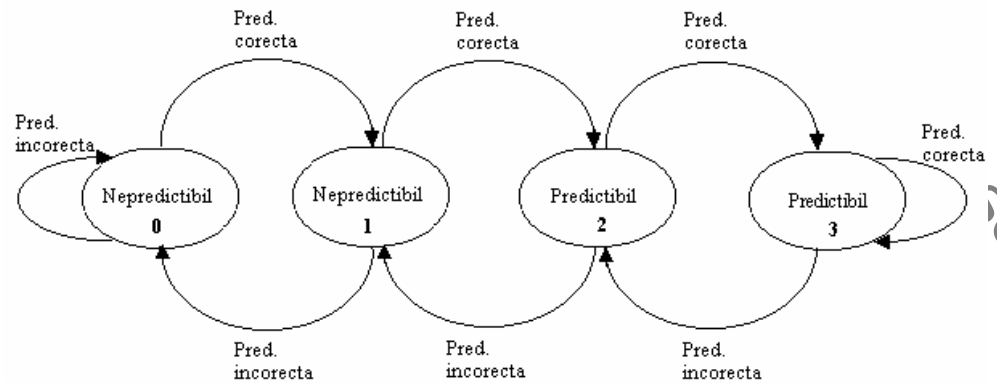


Figura 4.17. Diagrama de stări şi tranziţii – automat finit pe doi biţi (de tip numărător saturat)

Tabela de tranziție arată astfel:

Stare curentă	**Stare următoare**		**Predicție** (Acțiune de ieşire: predictibil=1 / nepredictibil=0)
	Pentru intrare = 0 (predicție **incorectă**)	Pentru intrare = 1 (predicție **corectă**)	
0	0	1	0
1	0	2	0
2	1	3	1
3	2	3	1

Tabelul 4.4. Tranzițiile automatului de predicție pe 2 biți

Cei doi biți pot codifica până la patru stări (cazul nostru 0÷3). Funcționarea automatului este de tip numărător saturat întrucât pentru o intrare 1, starea este incrementată doar dacă este mai mică strict decât starea maximă altfel rămâne la valoarea maximă (saturație). Pentru o intrare 0, starea este decrementată doar dacă este mai mare strict decât starea minimă altfel rămâne la valoarea minimă (saturație). Într-o implementare software care modelează funcționarea automatului finit (de predicție) acesta poate fi este descris printr-un şir de caractere cu un format mai special, ce prezintă atât numărul de stări, tranzițiile între stări cât şi predicția aferentă fiecărei stări. De exemplu, automatul anterior s-ar descrie prin şirul de caractere: „01021323:12”. Prima parte a şirului până la caracterul ‘:’ reprezintă tranzițiile pentru starea ‘0’ cu intrare 0, apoi cu intrare 1, apoi tranzițiile din ‘1’ pentru aceleaşi intrări, etc. Numărul 12 din a doua parte este “văzut” în binar “invers” sub forma “0011” şi reprezintă ieşirile asociate fiecărei stări în parte (stării ‘0’ îi este asociat cel mai puțin semnificativ bit, în acest caz bitul ‘0’, următorul bit lui ‘1’, bitul ‘0’ etc).

Rolul acestui automat este următorul: starea automatului va determina acțiunea de ieşire (predicția ramificației în program, adică urmarea ramurii cu *if* sau a celei cu *else*) Dacă acesta e 1 logic, atunci se prezice că saltul se va face, iar dacă e 0 logic, se prezice că saltul nu se va face. Evident că nu se poate şti în avans dacă predicția este corectă. Oricum, procesorul va considera că predicția este corectă şi va declanşa aducerea instrucțiunii următoare de pe ramura prezisă. Dacă predicția se dovedeşte a fi fost falsă se va iniția procesarea celeilalte ramuri de program. Totodată, automatul va tranzita într-o nouă stare conform figurii 4.17 sau a tabelului 4.4.

Într-un circuit digital, o **implementare hardware a unui AF** necesită un **registru** pentru a stoca variabilele de stare, un bloc de **logică combinațională** care determină tranziția de stare, şi un alt **bloc de logică combinațională** care determină ieşirea automatului finit [Wik]. **Optimizarea** unui automat finit înseamnă găsirea automatului finit cu numărul minim de stări care operează cu aceeaşi funcționalitate. Această problemă se poate rezolva folosind un **algoritm de colorare** [Wik, Cor90].

Automatele finite pot fi clasificate în: *Acceptoare* şi *Transductoare*. Automatele *acceptoare* generează o ieşire binară, fie *da*, fie *nu*, reprezentând răspunsul la întrebarea *"Intrarea este acceptată sau nu de maşină?"*. Maşina, utilizată în cazul gramaticilor din limbajele formale, poate fi descrisă şi ca definitorie pentru un limbaj, în cazul de față limbajul definit ar conține toate cuvintele acceptate de maşină şi nici unul din cele neacceptate. Toate stările automatului se clasifică în stări acceptante (finale) sau neacceptante. Dacă la momentul terminării procesării întregului şir de intrare automatul este într-o stare finală, atunci intrarea este acceptată, altfel nu. Ca o regulă, intrarea este compusă din simboluri (caractere); nu se folosesc acțiunile. *Transductoarele* generează ieşire pe baza unei intrări date şi/sau a unei stări, folosind acțiuni. Ele sunt folosite în controlul aplicațiilor. Aici se disting două tipuri [Wik]:

- Maşina *Moore* – Automatul foloseşte doar acțiuni de intrare, şi deci ieşirea depinde doar de stare. Avantajul modelului Moore este dat de simplificarea comportamentului.
- Maşina *Mealy* – Automatul foloseşte doar acțiuni de intrare de date, adică ieşirea depinde de intrare şi de starea curentă. Utilizarea unui AF Mealy conduce adesea la o reducere a numărului de stări.

În practică se folosesc deseori modele hibride. O altă distincție care se face între automatele finite este cea între *automatele finite deterministe* (AFD) şi cele *nedeterministe* (AFN). În cazul automatelor deterministe, din fiecare stare se poate efectua exact o singură tranziție pentru fiecare intrare posibilă. În cazul automatelor nedeterministe, pentru o anumită stare şi o anumită intrare, pot fi mai multe tranziții posibile, sau chiar nici una. Această distincție este relevantă în practică, dar nu şi în teorie, deoarece există un algoritm care poate transforma orice AFN într-un AFD echivalent, deşi această transformare măreşte, de obicei, complexitatea automatului.

Modelul matematic

În funcție de tip, există mai multe definiții. Un automat finit **acceptor** este un cvintuplu $<\Sigma, S, s_0, \delta, F>$, unde:

- Σ este alfabetul de intrare (o mulțime finită şi nevidă de simboluri).

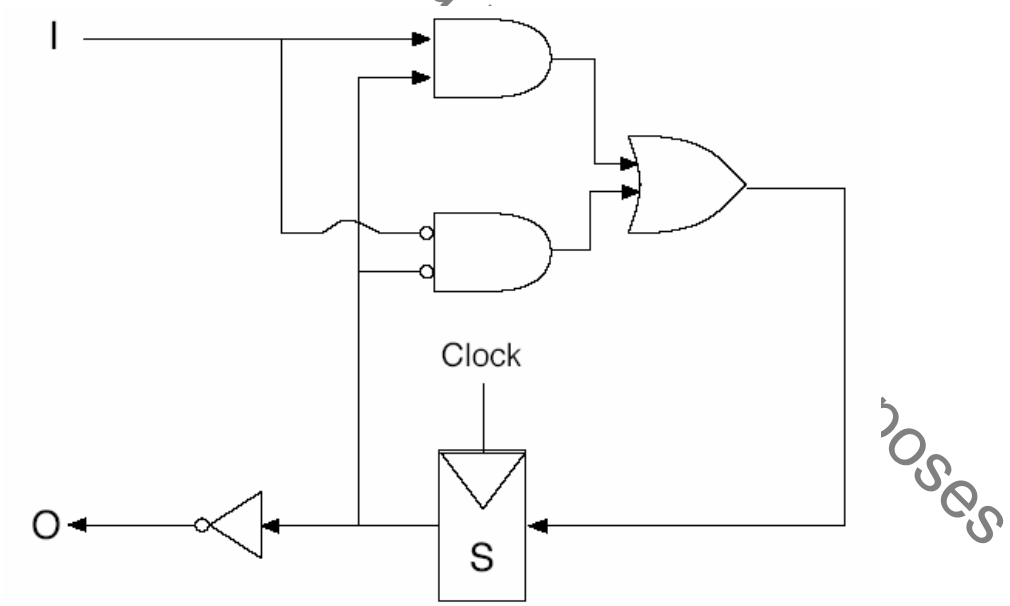
- S este o mulțime finită şi nevidă de stări.
- s_0 este starea inițială, element al lui S. Într-un automat finit nedeterminist, s_0 este o mulțime de stări inițiale.
- δ este funcția de tranziție a stării: δ: S x $\Sigma \rightarrow S$.
- F este mulțimea stărilor finale, o submulțime (posibil vidă) a lui S.

Un automat finit **transductor** este un sextuplu $<\Sigma, \Gamma, S, s_0, \delta, \omega>$, unde:

- Σ, S şi s_0 îşi păstrează semnificațiile din definiția automatului finit acceptor.
- Γ este alfabetul de ieşire (o mulțime finită şi nevidă de simboluri).
- δ este funcția de tranziție a stării: δ: S x $\Sigma \rightarrow S$ x Γ.
- ω este funcția de ieşire.

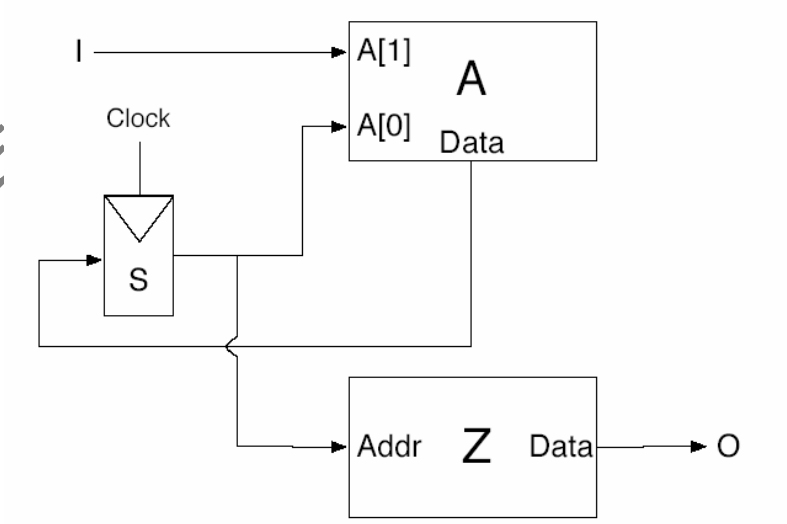
Dacă funcția de ieşire este o funcție de stare şi de alfabetul de intrare (ω: S x $\Sigma \rightarrow \Gamma$), atunci această definiție corespunde **modelului Mealy**. Dacă funcția de ieşire depinde doar de stare (ω: $S \rightarrow \Gamma$), atunci această definiție corespunde **modelului Moore**.

Aplicație 2: Circuitul de mai jos implementează un automat cu număr finit de stări (fie acesta M). Registrul pe un bit etichetat S memorează starea curentă, I este intrarea şi O este ieşirea.



(a) Câte stări are automatul M? Desenați diagrama stărilor şi a tranzițiilor pentru automatul M.

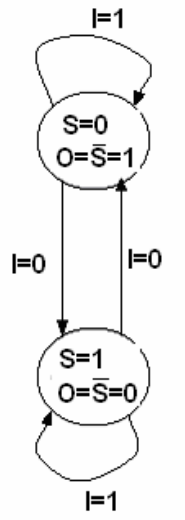
(b) Circuitul de mai jos este un automat cu număr finit de stări programabil. Componentele circuitului, sunt etichetate astfel: (A) o memorie cu patru locații fiecare conținând un bit (2^2x1-bit), (Z) o memorie cu două locații a câte un bit fiecare (2^1x1-bit). (S) un registru pe un bit.



Ce trebuie să conțină locațiile de memorie A şi Z astfel încât acest circuit să reprezinte o implementare hardware a automatului M?

Adresa (A_1A_0)	**Conținutul locației A**
00	
01	
10	
11	
Adresa (Z_0)	**Conținutul locației Z**
0	1
1	0

Răspuns:
a) Întrucât S este reprezentat pe un bit rezultă că M are două stări. Diagrama stărilor şi a tranzițiilor pentru automatul M arată astfel:



Considerăm o stare a automatului (S=0) şi alta (S=1). Din schemă se observă că O= **not** S. De asemenea, pentru orice intrare egală cu 0 (I=0) automatul trece din starea curentă în cealaltă (din S=0 în S=1 sau invers). Pentru orice intrare egală cu 1 (I=1) automatul îşi păstrează starea.

b)

Adresa (A_1A_0)	**Conținutul locației**
00	1
01	0
10	0
11	1

Se observă că A_1 este I (intrarea în automat) iar A_0 este S (starea curentă). Ţinem cont de considerațiile anterioare (intrare 1 păstrează starea constantă, intrare 0 schimbă starea).

Adresa (Z_0)	**Conținutul locației**
0	1
1	0

Se observă că O (ieşirea) este dat de conținutul locației, şi este negația stării S.

În finalul acestui curs, anticipând puțin ce va fi prezentat în cursurile următoare, este ilustrată diagrama fluxului de date din cadrul calculatorului LC-3. Aceasta conține toate structurile logice digitale, combinaționale şi secvențiale, care combinate îndeplinesc funcția de procesare a informațiilor,

funcție *cheie* a oricărui model de calcul de tip von Neumann. Deşi pare demnă de intimidat, schema este prezentată aici pentru a se observa că, în acest moment, se dispune de cunoştințe suficiente pentru înțelegerea funcționalității unui calculator.

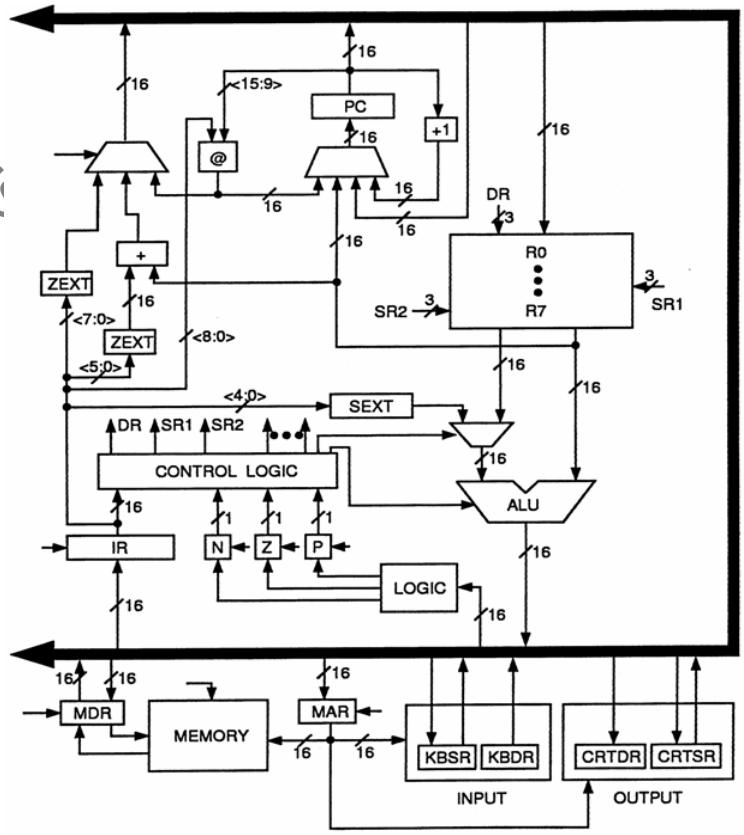
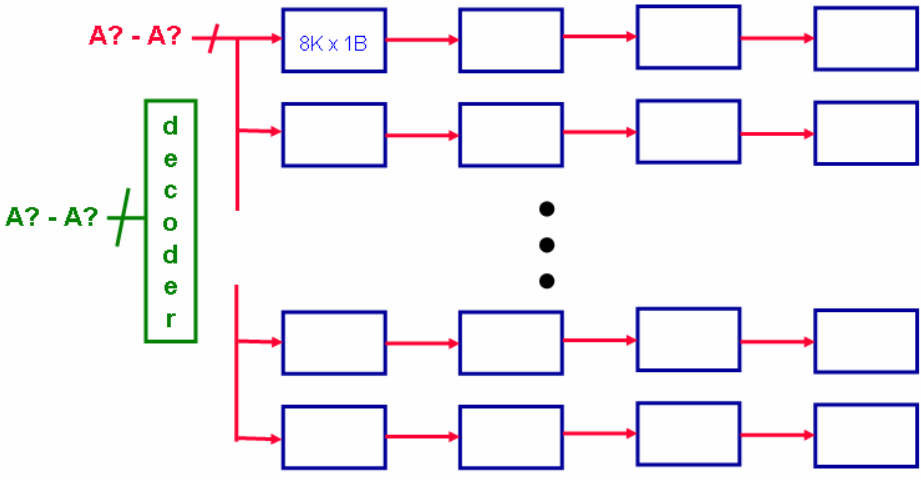


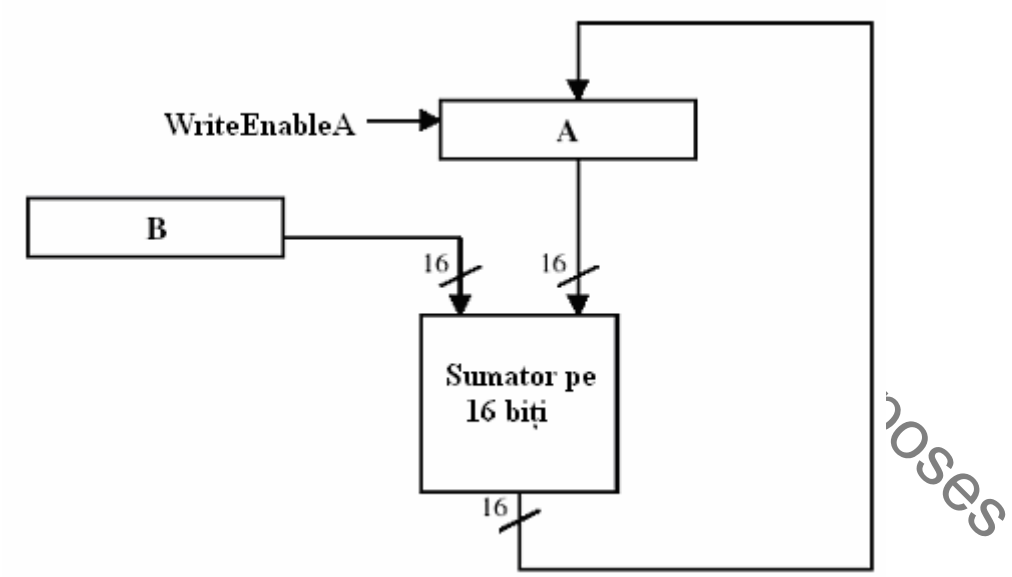
Figura 4.18. Fluxul datelor la calculatorul LC-3

4.3. EXERCIȚII ŞI PROBLEME

1. Să se implementeze folosind circuite primare de memorie de capacitate **8Klocații x 1 octet** o memorie de **64Kcuvinte**, fiecare **cuvânt** având **32 biți**. Care sunt liniile de adresă dacă memoria este adresată pe cuvânt ? Dar dacă este adresată pe octet ?

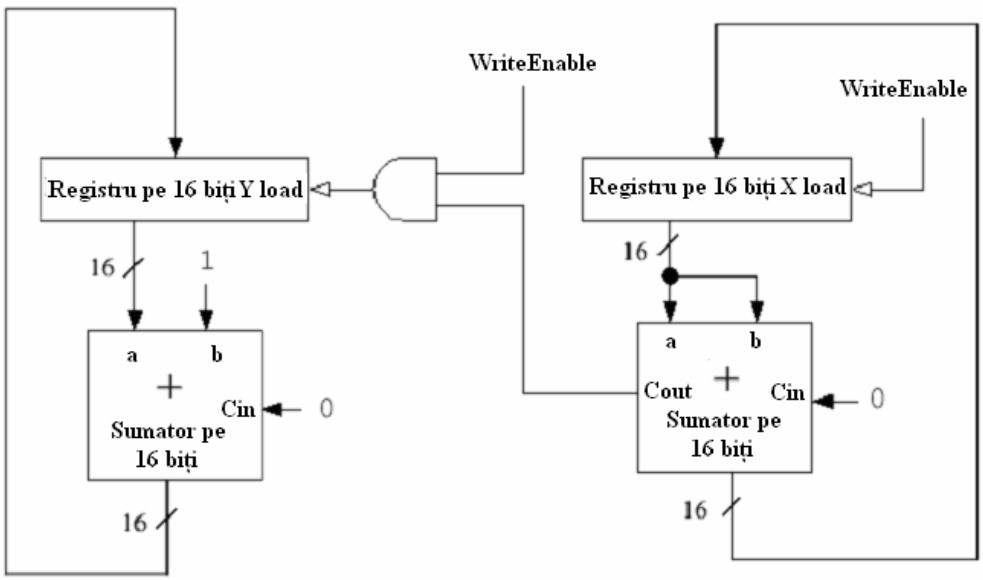


2. În diagrama logică de mai jos **A** şi **B** sunt doi regiştrii pe 16 biţi conectaţi la un sumator care realizează operaţii de adunare pe 16 biţi. Se presupune că registrul **A** are valoarea iniţială 0 şi registrul **B** are valoarea *X*. Ce valoare va conţine registrul A dacă semnalul **WriteEnableA** este activat de 5 ori ? Dar dacă **WriteEnableA** este activat de *N* ori ?

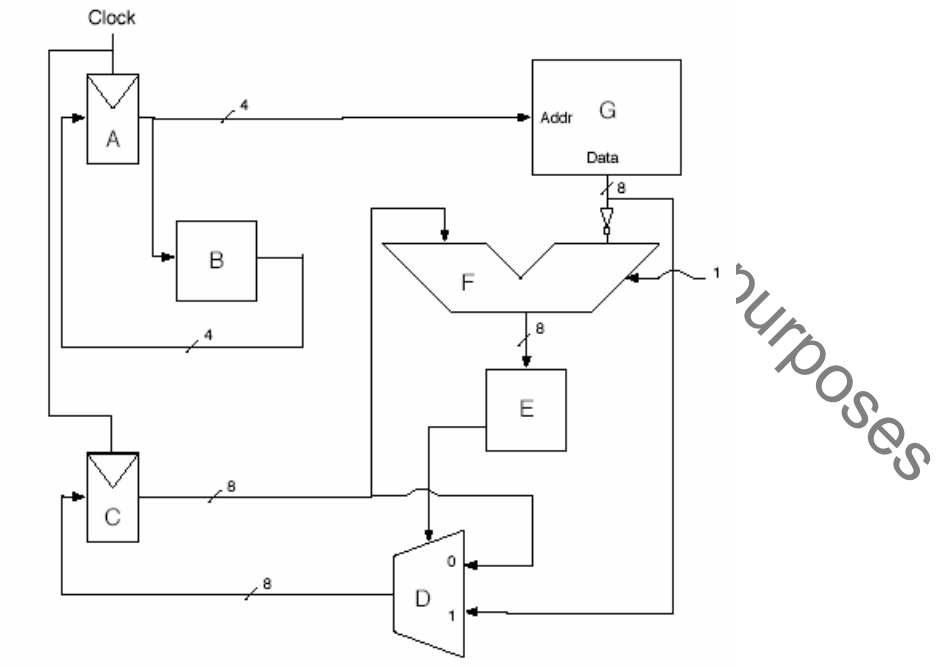


3. În diagrama logică de mai jos se presupune că Y conţine iniţial valoarea 0 în timp ce X conţine un număr oarecare în complement faţă de 2. Într-

o singură propoziţie descrieţi ce valoare va reţine Y după ce semnalul **WriteEn** este ridicat în ‘1’ logic şi coborât în ‘0’ logic de 16 ori.



4. **Automate secvenţiale si programabile:** Se consideră diagrama de funcţionare a următorului circuit.



Componentele circuitului, sunt etichetate astfel: (A) un registru pe 4 biți reprezentând un număr întreg fără semn (inițializat cu valoarea 0); (B) un numărător pe 4 biți; (C) un registru pe 8 biți care memorează un întreg în complement față de 2 (inițial setat pe 127); (D) un multiplexor pe 8 biți 2-la-1; (E) un circuit al cărui ieşire este *true* (1) atunci când intrarea (un întreg în complement față de 2 pe 8 biți) este pozitiv; (F) un sumator pe 8 biți; (G) o memorie cu 16 locații fiecare conținând cuvinte de 8 biți (2^4x8-bit). Se presupune că această memorie are următorul conținut:

Adresa	Conținut	Adresa	Conținut
0	31	8	27
1	47	9	95
2	5	10	88
3	16	11	67
4	112	12	63
5	3	13	54
6	59	14	80
7	8	15	110

(a) Determinați valorile stocate în regiştrii A şi C la sfârşitul fiecărui ciclu de tact pentru 8 ciclii desfăşurați. Completați următorul tabel cu informațiile lipsă.

Ciclu de tact	**Registrul A**	**Registrul C**
0	0	127
1		
2		
3		
4		
5		
6		
7		
8		

(b) Într-o singură propoziție, ce realizează circuitul ?

5. MODELUL ARHITECTURAL *VON NEUMANN*. COMPONENTE DE BAZĂ. PRINCIPIILE PROCESĂRII INSTRUCȚIUNILOR

5.1. COMPONENTELE DE BAZĂ ALE MODELULUI ARHITECTURAL VON NEUMANN

În anii '40, matematicianul **John von Neumann** [Wik] analizează starea de fapt a calculatoarelor şi scrie în **1945** un raport intitulat „*First Draft of a Report on the EDVAC*" (Prima schiță a unui raport despre **EDVAC – Electronic Discrete Variable Automatic Computer**), în care sugerează o arhitectura revoluționară care să transpună în realitate (implementează fizic) maşina universală de calcul Turing. În această arhitectură, programul nu mai este reprezentat de felul în care sunt cuplate unitățile funcționale, ci este stocat în memorie, fiind descris folosind un limbaj numit cod-maşină. În cod-maşină, operațiile de executat sunt codificate sub forma unor numere numite *instrucțiuni*. El a recomandat sistemul binar pentru memorarea datelor şi a propus instrucțiuni de control a acestora. Programul de executat este descris printr-un şir de instrucțiuni, care se execută consecutiv. Von Neumann a introdus ideea grupării instrucțiunilor în subrutine care să fie apelate repetat. Subrutinele des folosite nu trebuiau reprogramate pentru fiecare nou program ci puteau fi păstrate în *biblioteci* şi citite din memorie atunci când erau necesare. Pe lângă unitățile funcționale care execută operații aritmetice, calculatorul mai are o unitate de control, care *citeşte secvențial instrucțiunile programului şi care trimite semnale între unitățile funcționale pentru a executa aceste instrucțiuni*. *Rezultatele intermediare sunt stocate în memorie*. Această arhitectură se numeşte „**von Neumann**". Marea majoritate a calculatoarelor din ziua de azi sunt bazate pe această arhitectură; noțiunea de limbaj-maşină, şi cea înrudită, de limbaj de programare, folosite pentru descrierea programelor, sunt concepte foarte naturale pentru toți cei care manipulează calculatoarele. Von Neumann propune **calculatorul** să fie văzut ca sistem de procesare a informației (bazat pe procesor – CPU), adică un **mecanism care direcționează dar şi realizează procesarea informației**

(interacționează cu instrucțiuni şi date preluate de la intrare şi produce rezultate la ieşire).

Cunoştințele anterior dobândite (o bogată colecție de circuite logice combinaționale şi secvențiale) vor permite implementarea şi funcționarea logico-fizică a modelului arhitectural von Neumann. Figura 5.1 ilustrează structura de bază a calculatorului (arhitecturii de procesare) propusă de von Neumann pentru execuția programelor.

Se disting cinci componente principale:

- **Memoria**: reține atât instrucțiuni cât şi date.
- **Unitatea de procesare**: execută instrucțiunile din programe.
- **Unitatea de control**: parcurge secvențial şi interpretează instrucțiunile, controlând fluxul de execuție al programelor.
- **Unitatea de intrare date** (*Input*): permite introducerea datelor din exterior în memoria calculatorului.
- **Unitatea de ieşire** (*Output*): produce rezultatul generat de program pentru utilizator.

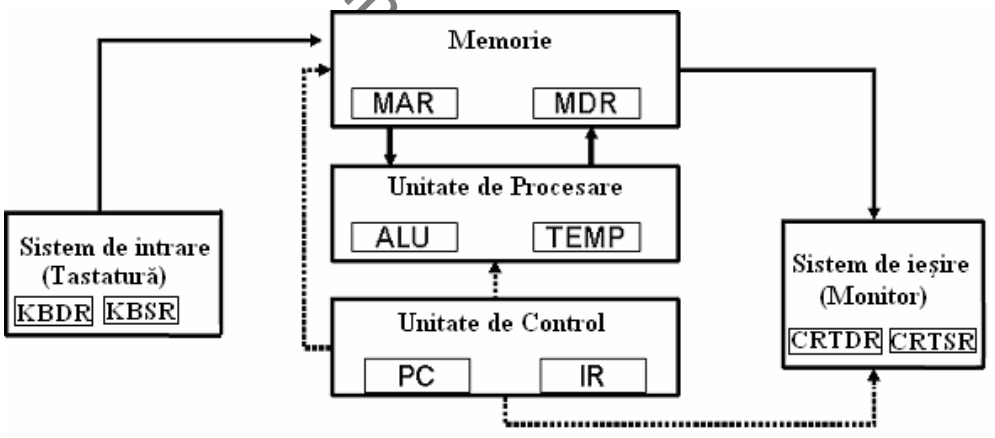


Figura 5.1. Modelul arhitectural *von Neumann*

Memoria are două caracteristici importante: **capacitatea** (este de dorit a fi cât mai mare) şi **timpul de acces** (este de dorit a fi cât mai mic), parametrii cu tendințe antagoniste. Practic, cu cât capacitatea memoriei crește, cu atât scade timpul de acces la memorie şi se ieftinește prețul per unitate de octet memorat. Creşterea decalajului dintre viteza procesoarelor şi timpul de acces la memorie impune introducerea unui sistem ierarhic de memorie, pentru a nu face simțită la nivelul performanței globale a sistemului încetineala cu care se accesează memoria [Vin00].

Fiecare locație de **memorie** este caracterizată de *adresă* şi *conținut*. **Adresa** reprezintă o secvență binară (*pattern* de biți) care identifică **în mod**

unic o locaţie de memorie. **Conţinutul** reprezintă valoarea (*pattern*ul de biţi) stocat la respectiva adresă. Numărul total de locaţii de memorie disponibile formează *spaţiul de adrese*. Calculatoarele LC-2 şi LC-3 dispun de un spaţiu de adrese de 2^{16} locaţii (magistrala de adrese este pe 16 biţi). Uzual, calculatoarele actuale sunt dotate cu o memorie principală (centrală) de tip RAM de capacitate 1 GByte (2^{32} adrese, la fiecare adresă existând un cuvânt de 8 biţi). **Adresabilitatea** constituie numărul de octeţi de date (cuvânt de memorie – *word*) disponibili la o anumită adresă şi utilizaţi de către unitatea de procesare. În mod frecvent, o instrucţiune trebuie să scrie sau să citească un întreg cuvânt (word) la fiecare acces în memorie. Accesarea memoriei se realizează prin intermediul celor doi regiştrii MAR şi MDR. MAR (registrul de adresă al memoriei) este încărcat cu valoarea adresei de la care sau la care se va citi sau scrie din / în memorie. Valoarea din MAR va fi depusă pe decodificatorul de adrese al memoriei. Pentru instrucţiunile care citesc date din memorie (de tip *load*) conţinutul memoriei se încarcă în MDR (registrul de date al memoriei). Pentru instrucţiunile de scriere în memorie (de tip *store*), în MDR se înscrie valoarea ce va fi stocată în memorie la activarea semnalului *WriteEnable*.

Pentru o mai bună înţelegere a noţiunilor de adresă de memorie şi respectiv conţinutul memoriei, se consideră următorul exemplu: *fie o memorie de tipul* (2^3x3-biţi) *a cărei locaţii sunt numerotate de la adresa* 000_2 *la* 111_2. *În această memorie la adresa* 110_2 *se află cuvântul* 000_2 *iar la adresa* 000_2 *se află cuvântul de date* 001_2. Se reaminteşte că, adresele sunt unice (nu pot exista două locaţii distincte caracterizate de aceeaşi adresă). Însă, conţinuturile a două sau mai multe locaţii pot coincide.

Adresa	**Conţinut**
000_2	001_2
001_2	
010_2	
011_2	
100_2	
101_2	
110_2	000_2
111_2	

Unitatea de procesare reprezintă motorul de execuţie al programului. Este compusă din mai multe unităţi, fiecare îndeplinind o anumită funcţie complexă (de la adunări, scăderi la înmulţiri, împărţiri, extrageri de rădăcină pătrată, etc). O unitate de procesare minimală conţine o **unitate aritmetico-**

logică (ALU) şi un **set de regiştrii de uz general**. Numărul de biți ai datei cu care operează unitatea de procesare se numeşte dimensiunea cuvântului de date (*word size*) al maşinii (arhitecturii). ALU realizează operațiile aritmetico-logice de bază (adunare / scădere / AND / NOT) în general pe întregul cuvânt de date (word) dar uneori şi pe subdiviziuni ale acestuia (octet, bit). Procesorul MIPS are cuvântul de date pe 32 de biți dar instrucțiunile acestuia pot opera şi pe octet (byte), semicuvânt (*halfword*). Cuvântul de date la LC-2 este pe 16 biți, la Intel Pentium IV pe 32 de biți, la Intel Itanium, PowerPC şi Alpha Compaq pe 64 de biți. Procesoarele din sistemele dedicate (telefoane celulare, maşini de spălat, frigidere, copiatoare, imprimante, etc) necesită cuvinte de date pe doar 8 biți. Valorile cu care operează unitatea aritmetico-logică sunt stocate în zone de memorie de capacitate foarte redusă şi caracterizate de viteză de operare (citire/scriere/deplasare/rotire) foarte ridicată – regiştrii procesorului. LC-2 deține 8 regiştrii (R_0-R_7) fiecare pe 16 biți, arhitectura Alpha ISA deține 32 de regiştrii fiecare pe 64 de biți, procesoarele MIPS R2000, R3000 dețin 32 de regiştrii pe 32 de biți fiecare (MIPS R16000 are 64 regiştrii care pot lucra cu date atât pe 64 cât şi pe 32 de biți).

Unitatea de control coordonează toate acțiunile necesare pentru execuția instrucțiunilor: **extrage** (*instruction fetch*) din memorie (sau din cache-ul de instrucțiuni) instrucțiunea, o decodifică (*decode*), preia operanzii corespunzători din regiştrii sau memorie (*operand fetch*) şi o execută (*execute*). Foloseşte doi regiştrii de interfață cu memoria: *PC* (*program counter*) – în alte arhitecturi poartă denumirea de *instruction pointer*, reține adresa următoarei instrucțiuni care va fi adusă din memorie şi *IR* (*instruction register*) păstrează instrucțiunea curentă care se execută.

Unitățile de intrare / ieşire sunt cunoscute sub numele generic de dispozitive **periferice** nu din cauză că sunt mai puțin importante pentru procesor ci datorită poziției acestor dispozitive față de CPU. În cazul calculatorului (simulatorului) LC-3 (LC-2) dispozitivul de intrare îl reprezintă tastatura cu regiştrii de interfață (KBDR – de date şi KBSR – de stare) iar dispozitivul de ieşire este monitorul cu regiştrii de interfață (CRTDR – de date şi CRTSR – de stare). Evident că există o varietate de dispozitive de intrare (mouse, scanere digitale, stick-uri USB, dischete, etc) şi de ieşire (imprimante, ecrane cu cristale lichide, harddisk).

5.2. PRINCIPIILE PROCESĂRII INSTRUCȚIUNILOR

Ideea centrală a modelului de procesare propus de *von Neumann* evidențiază faptul că atât instrucțiunile cât şi datele sunt memorate ca şi secvențe de biți, în memoria calculatorului, programul fiind executat succesiv (*pas–cu–pas*) câte o instrucțiune la un moment dat în funcție de direcția indicată de unitatea de control.

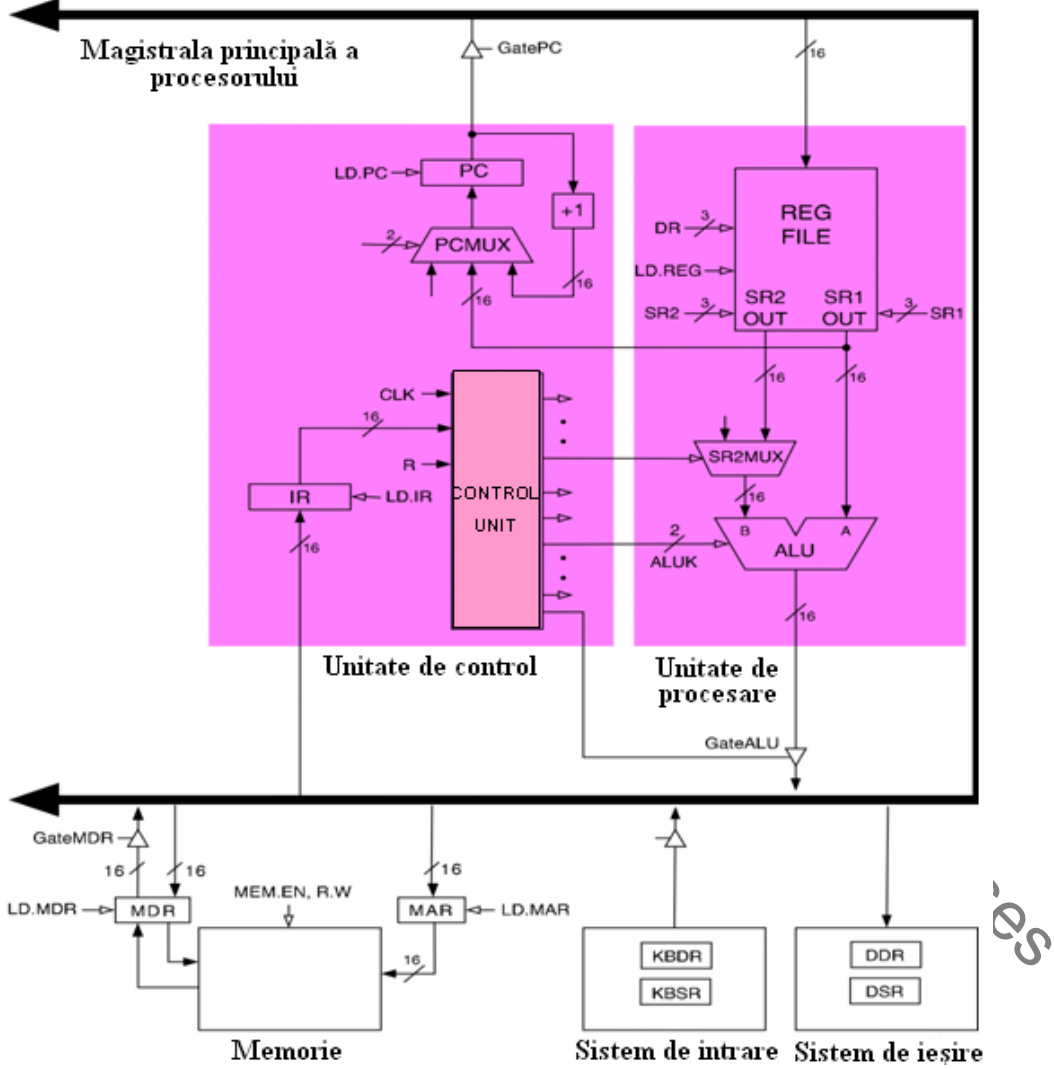


Figura 5.2. Calculatorul LC-3 văzut ca un model de procesare *von Neumann* [Pal07]

După cum se poate observa în figura 5.2, nucleul central („*inima*") al unității de control îl reprezintă un automat cu număr finit de stări care stabileşte starea şi ieşirea automatului şi a circuitelor logice conform cu fiecare tip de instrucțiune (selecția regiştrilor sursă/destinație, tipul operației: cu registru sursă sau cu valoare imediată, tipul instrucțiunii: de salt necondiționat sau apel de subrutină, etc). Procesul este controlat de către ceasul sistemului. Automatul cu număr finit de stări efectuează câte o tranziție („*ciclu maşină*") la fiecare impuls de tact (perioada de tact este egală cu inversul frecvenței procesorului $T_{CLK} = \frac{1}{f_{CLK}}$).

5.2.1. CICLUL INSTRUCȚIUNII

Văzută din exterior, instrucțiunea reprezintă unitatea atomică (fundamentală) a unui program. Cu toate acestea instrucțiunea este divizată pe câmpuri de către unitatea de control şi interpretată permițându-se apoi execuția ei. În cadrul procesoarelor RISC (MIPS, Alpha, PowerPC) formatul instrucțiunilor este de lungime fixă (uzual 32 de biți). Fiecare instrucțiune este caracterizată de două câmpuri de bază:

- **Codul operației** *(opcode)***:** specifică operația ce se va executa.
- **Operanzii:** sursa respectiv destinația instrucțiunii.

Setul de instrucțiuni al calculatorului, formatul acestora, setul de regiştri, tipurile de date utilizate şi modurile de adresare (mecanismul prin care calculatorul / procesorul localizează operanzii) constituie arhitectura setului de instrucțiuni (ISA).

Succesiunea de etape prin care trece instrucțiunea de la aducerea ei din memorie până la încheierea execuției sale se numeşte **ciclul instrucțiunii**. Von Neumann a propus şase etape (faze de procesare ale instrucțiunii):

- **Fetch**: aduce instrucțiunea din memorie de la adresa dată de registrul PC şi o depune în registrul instrucțiunii (IR). Chiar şi această etapă poate fi divizată în subetape:
  - (1) Conținutul registrului PC se depune în MAR (*MAR ← (PC)*).
  - (2) Automat PC-ul este incrementat pentru a pointa spre adresa următoarei instrucțiuni (*PC ← (PC) + 1*).
  - (3) Se citeşte instrucțiunea din memorie odată cu accesarea semnalului READ (*MDR ← Mem[MAR]*).
  - (4) Instrucțiunea se depune din registrul de date al memoriei în IR (*IR ← (MDR)*).

Dacă subetapele (1), (2) şi (4) necesită un singur ciclu de tact (operaţii foarte rapide de transfer sau adunare), subetapa (3) poate dura mai mult în funcţie de decalajul în timp dintre timpul de acces la memorie şi viteza procesorului.

- **Decode**: este identificată operaţia ce va avea loc şi sunt stabiliţi operanzii (regiştrii sursă şi cel destinaţie). Opcode-ul este depus pe intrarea unui decodificator care va decide secvenţa de evenimente solicitate de instrucţiunea în cauză.
- **Evaluate address**: în cazul instrucţiunilor cu operand în memorie este calculată adresa acestuia (cunoscută sub numele de adresă efectivă – *effective address*).
- **Fetch operands**: sunt determinaţi operanzii sursă pentru execuţia instrucţiunilor – fie din setul de regiştrii generali ai procesorului fie din memorie. Regiştrii sursă vor deveni operanzi de intrare pentru unitatea aritmetico-logică sau pentru memorie. Pentru operandul stocat în memorie este folosită adresa efectivă (EA), calculată la pasul anterior (evaluate address).
- **Execute**: se execută operaţia codificată în instrucţiune. De exemplu, pentru instrucţiunile aritmetico-logice se adună/scade etc., cei doi operanzi (registru cu registru sau registru cu valoare imediată). Dacă instrucţiunea este de ramificaţie (modifică fluxul de control al programului) atunci registru PC este actualizat cu o nouă valoare – adresa destinaţie a instrucţiunii de salt. Pentru instrucţiunile de transfer cu memoria (*load / store*) în această fază nu se face nimic.
- **Store results**: scrie rezultatul instrucţiunii în destinaţia corespunzătoare (în registru sau în memorie). Adresa efectivă anterior calculată va indica de unde se aduce rezultatul sau unde se scrie în memorie.

O observaţie care rezultă se referă la faptul că nu toate instrucţiunile necesită toate cele 6 faze de procesare. Pentru instrucţiunile care folosesc doar regiştrii (formatul R-tip la instrucţiunile procesoarelor RISC) faza *Evaluate Address* poate fi sărită (*skip*). Pentru instrucţiunile cu referire la memorie (formatul I-tip la instrucţiunile procesoarelor RISC cu mod de adresare indirect registru şi nu indexat) poate fi evitată faza Execute. De asemenea, nu toate fazele de procesare necesită acelaşi număr de perioade de tact.

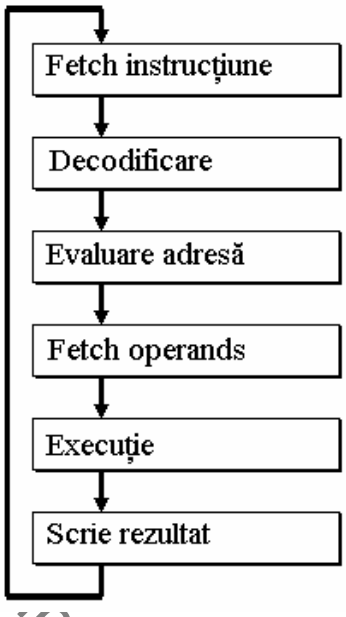


Figura 5.3 Ciclul instrucțiunii în concepția lui *von Neumann*

După cum se poate observa în figura 5.3 unitatea de control repetă ciclul instrucțiunii pentru toate instrucțiunile din program, ghidată de valoarea registrului PC. Întrucât valoarea acestuia este incrementată pe durata fazei *fetch* sau este actualizată de instrucțiunile de ramificație şi control pe durata fazei *execute* rezultă că în fiecare moment se cunoaşte adresa următoarei instrucțiuni. La fel ca şi în cazul structurilor de date cu legături (liste, arbori), cunoscând adresa primei instrucțiuni se poate trece la execuția întregului program.

Deşi nu toate tipurile de instrucțiuni necesită toate cele şase faze de procesare şi, în ciuda tehnicilor avansate de execuție din cadrul procesoarelor *pipeline*[4] *superscalare* care presupun până la 20 şi chiar 31 de nivele pipeline de procesare (vezi Intel Pentium 4 versiunile Northwood şi Prescott), fazele de procesare propuse de von Neumann (v.N) au rămas

[4] Tehnica de procesare *pipeline* reprezintă o tehnică de procesare paralelă a informației prin care un proces secvențial este divizat în subprocese, fiecare subproces fiind executat într-un segment special dedicat şi care operează în paralel cu celelalte segmente. Fiecare segment execută o procesare parțială a informației. Rezultatul obținut în segmentul *i* este transmis în tactul următor spre procesare segmentului (*i*+1). Rezultatul final este obținut numai după ce informația a parcurs toate segmentele, la ieşirea ultimului segment. Denumirea de pipeline provine de la analogia cu o bandă industrială de asamblare. Este caracteristic acestor tehnici faptul că diversele procese se pot afla în diferite faze de prelucrare în cadrul diverselor segmente, simultan.

actuale şi în ziua de azi. Din punct de vedere al performanţei sistemelor de calcul, evoluţia de la modelul de procesare propus de von Neumann la cele actuale constă în următorul fapt: modelul iniţial (v.N) presupunea că instrucţiunile sunt executate secvenţial, una câte una; doar după ce o instrucţiune aflată în curs de execuţie a efectuat faza *store result* o alta poate efectua faza *fetch*. Procesoarele moderne presupun un paralelism atât temporal de tip pipeline (mai multe faze de procesare aferente unor instrucţiuni diferite executate în acelaşi timp) cât şi unul spaţial de tip *superscalar* (mai multe unităţi de execuţie aferente aceleaşi faze de procesare).

5.2.2. TIPURI DE INSTRUCŢIUNI

Calculatorul (simulatorul) LC-3 deţine un set de 16 instrucţiuni, astfel încât câmpul *opcode* este codificat pe 4 biţi. De asemenea, zona temporară de stocare din cadrul unităţii de procesare (regiştrii procesorului) constă din 8 locaţii (R0-R7). Codificarea operanzilor (sursă, destinaţie) în corpul instrucţii se face pe 3 biţi per operand. Se disting trei mari categorii de instrucţiuni [Patt03]:

- *Operaţionale* (instrucţiuni de procesare a datelor – aritmetico-logice)

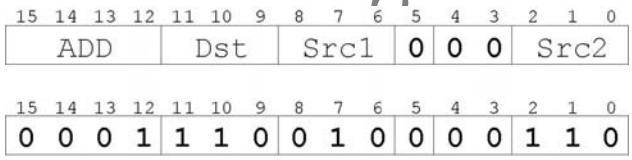


Figura 5.4. Exemplu de instrucţiune de adunare la calculatorul LC-3

Semantica instrucţiunii din figura 5.4 este următoarea: ADD R6, R2, R6 sau *"Adună conţinutul registrului R2 la conţinutul registrului R6 şi stochează rezultatul în registrul R6."*

- *Instrucţiuni de transfer date (transfer de date între memorie şi regiştrii procesorului)*

Pentru exemplificare se consideră instrucţiunea LDR (încarcă o dată de la o locaţie de memorie într-un registru al procesorului). Adresa de memorie se determină printr-o adunare dintre un registru de bază şi un deplasament (*offset*). Este întâlnită în cazul instrucţiunilor din programele de nivel înalt care prelucrează structuri de date omogene (vectori, matrici).

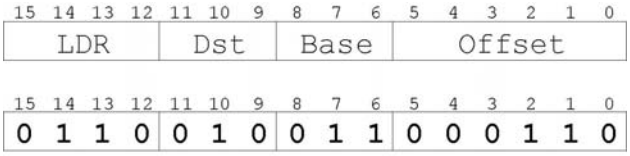


Figura 5.5. Exemplu de instrucțiune de aducere din memorie la calculatorul LC-3

Semantica instrucțiunii din figura 5.5 este următoarea: LDR R2, 6(R3) sau *"Adună valoarea 6 la conținutul registrului R3 pentru a forma adresa de memorie. Încarcă data din memorie de la adresa calculată în registrul R2."*

- De *Control* (Instrucțiuni de ramificație sau modificare a fluxului de execuție al programului). În cadrul acestei categorii se încadrează instrucțiunile de salt condiționat, necondiționat, indirecte, de apel de subrutine.

Teorema lui Böhm-Jacopini afirmă că orice program (de nivel înalt) poate fi descris folosind structuri secvențiale, alternative şi repetitive. Astfel, dacă nu apare în cadrul programului nici o structură alternativă (*if/else, switch/case*) sau repetitivă (*repeat, while, for*) în faza FETCH unitatea de control aduce câte o instrucțiune de la adresa dată de PC şi îi incrementează apoi valoarea acestuia. Instrucțiunile de ramificație din programele asamblare poartă numele de *jump*-uri (*salturi*) sau *branch*-uri. Jump-urile sunt necondiționate (modifică întotdeauna PC-ul) iar branch-urile sunt condiționate (modifică PC-ul doar dacă o condiție logică este îndeplinită, spre exemplu dacă conținutul unui registru este egal cu 0). Pentru aceste instrucțiuni pe durata fazei EXECUTE este actualizat PC-ul cu adresa destinație a saltului.

Pentru exemplificare se consideră instrucțiunea JMPR (încarcă PC-ul cu valoarea obținută prin adăugarea unui *offset* la conținutul unui registru al procesorului. Valoarea care se încarcă în PC reprezintă adresa instrucțiunii de la care se continuă procesarea).

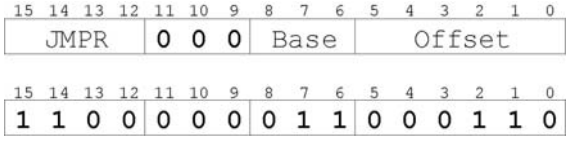


Figura 5.6. Exemplu de instrucțiune de salt necondiționat la calculatorul LC-3

Semantica instrucțiunii din figura 5.6 este următoarea: JMPR R3, #6 sau *"Adună valoarea 6 la conținutul registrului R3 şi încarcă rezultatul obținut în registrul PC."*

5.2.3. CEASUL PROCESORULUI

Motorul întregii activităţi de procesare a instrucţiunilor îl reprezintă semnalul de ceas. Ceasul sistemului este semnalul care „ţine unitatea de control în mişcare”. La fiecare impuls de tact (ceas) unitatea de control tranzitează la următorul ciclu maşină (următoarea instrucţiune sau următoarea fază din instrucţiunea curentă) – vezi figura 5.7.

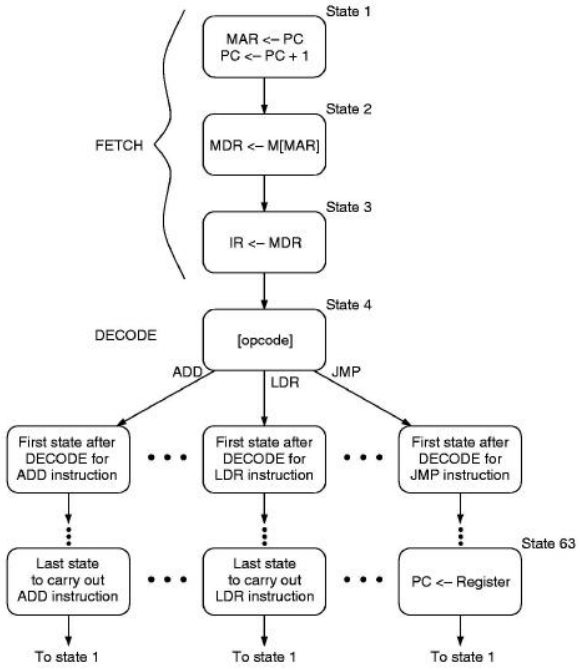


Figura 5.7. Ciclul instrucţiunii văzut ca un automat cu număr finit de stări

Circuitul generator de semnal de tact este bazat pe un cristal de cuarţ oscilator, care generează secvenţe regulate de nivele logice „0” şi „1”. Perioada de tact se determină între două fronturi ascendente ale semnalului de ceas (vezi figura 5.8b).

Oprirea sistemului de calcul

Programele utilizator se încheie prin transferarea controlului sistemului de operare (cea mai importantă aplicaţie software care rulează pe un sistem de calcul şi care are printre funcţiile sale pe cea de gestionare a procesorului şi a celorlalte resurse hardware). În continuare, sistemul de operare „intră într-o buclă de aşteptare” până când este lansată o nouă

aplicație utilizator. În tot acest timp, unitatea de control este activă şi parcurge ciclul instrucțiunii aferent instrucțiunilor componente programelor utilizator sau a celor aparținând sistemului de operare. Oprirea sistemului de calcul presupune oprirea unității de control, deci anularea semnalului de tact care reprezintă „*pulsul*" sistemului. Acest lucru se realizează printr-o operație de *SI logic* cu un generator de semnal „0"- logic (vezi figura 5.8a. Generatorul de semnal „0" este un bistabil de tip R-S pe a cărui intrări R şi S se setează valorile 0 şi 1 (R=0 şi S=1).

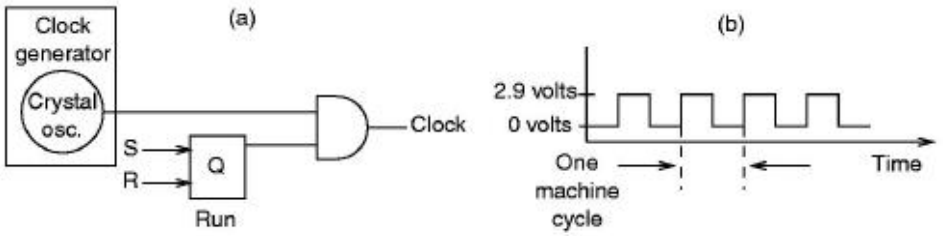


Figura 5.8. Oprirea sistemului de calcul

5.3. EXERCIȚII ŞI PROBLEME

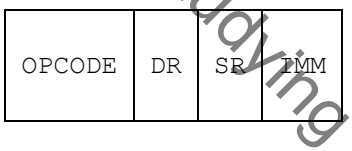
1. Pentru această problemă, se consideră o memorie cu 8 locații, având cuvântul de date pe 3 biți (2^3x3 biți), aşa cum se poate observa mai jos. Conținutul memoriei este următorul:

101	000
001	001
011	010
111	011
110	100
010	101
000	110
100	111

a) Ştiind că fiecare bit de memorie este reţinut folosind un bistabil de tip D, câţi astfel de bistabili sunt necesari pentru implementarea memoriei respective?

b) Presupunând că se începe citirea din memorie la locaţia **000** şi că fiecare valoare citită din memorie de la adresa curentă reprezintă adresa următoarei locaţii de unde se va citi, indicaţi şirul primelor 8 valori citite din memorie.

2. Se consideră următorul format de instrucţiune pe 32 de biţi, alcătuit din *OPCODE*, câmp pentru codificarea registrului destinaţie (*DR*), câmp pentru codificarea registrului sursă (*SR*) şi un câmp pentru valoarea imediată (*IMM*).



Dacă sunt implementate 60 de coduri de operaţie (opcodes) şi 16 regiştrii, care este cel mai mare număr pozitiv care poate fi reprezentat în câmpul IMM? (Se presupune că IMM este un număr în complement faţă de 2).

3. Se consideră o arhitectură de procesare care urmează modelul propus de Von Neumann (în care se specifică faptul că fiecare instrucţiune trebuie să-şi încheie execuţia (*store result*) înainte de aducerea altei instrucţiuni din memorie (*fetch instruction*)). Presupunând că, în medie o instrucţiune necesită 1.2 ciclii de tact pentru faza FETCH, 1 ciclu pentru decodificare (DECODE), 1 ciclu pentru evaluarea adresei operanzilor (EVALUATE ADDRESS), 1.8 ciclii de tact pentru aducerea operanzilor (FETCH OPERANDS), 2 ciclii de tact pentru faza EXECUTE şi 1 ciclu pentru scrierea rezultatului în setul de regiştrii ai procesorului (STORE RESULT). Câte instrucţiuni sunt executate într-o secundă de un procesor care rulează la o frecvenţă de 2.5 GHz ?

4. Se cunoaşte că frecvenţa unui calculator (procesor) este invers proporţională cu perioada sa de tact. Altfel spus, Frecvenţa=1/Perioada_de_tact. De exemplu, un procesor la 1 GHz are perioada de tact de 1 ns (10^{-9} s). Considerăm trei tipuri mari de instrucţiuni:

cu acces la memorie (load/store), aritmetico-logice (Add/And/Or, etc) şi altele (grupa instrucțiunilor de ramificație). O instrucțiune poate necesita mai mulți ciclii pentru execuție. Astfel, instrucțiunile cu referire la memorie durează 12 ciclii de tact, cele aritmetico-logice 9 ciclii de tact, în timp ce cele de ramificație necesită 10 ciclii de tact. Frecvența medie de apariție pe tipuri de instrucțiuni este ilustrată în următorul tabel:

Tipul Instrucțiunii	**Frecvența de apariție [%]**
Instrucțiuni cu referire la memorie	40%
Instrucțiuni aritmetico-logice	40%
Alte instrucțiuni	20%

Determinați cât timp (*s*) durează execuția unui program având 650.000 de instrucțiuni pe un calculator ce rulează la o frecvență de 3.2 GHz.

5. Următorul tabel reprezintă o mică memorie. Pentru următoarele întrebări se va face referire la acest tabel.

Adresa	**Data**
0000	x1E43
0001	xF025
0010	x6F01
0011	x0000
0100	x0065
0101	x0006
0110	xFED3
0111	x06D9

a) Care este valoarea binară conținută în locația 3? Dar în locația 6?
b) Valoarea binară conținută în fiecare locație de memorie poate fi interpretată în mai multe moduri: număr întreg în complement față de 2, număr flotant simplă sau dublă precizie, etc.
   b1) Interpretați locația 0 şi locația 1 ca întregi cu semn în complement față de 2.
   b2) Considerând locația 4 ca fiind un cod ASCII, despre ce caracter este vorba ?

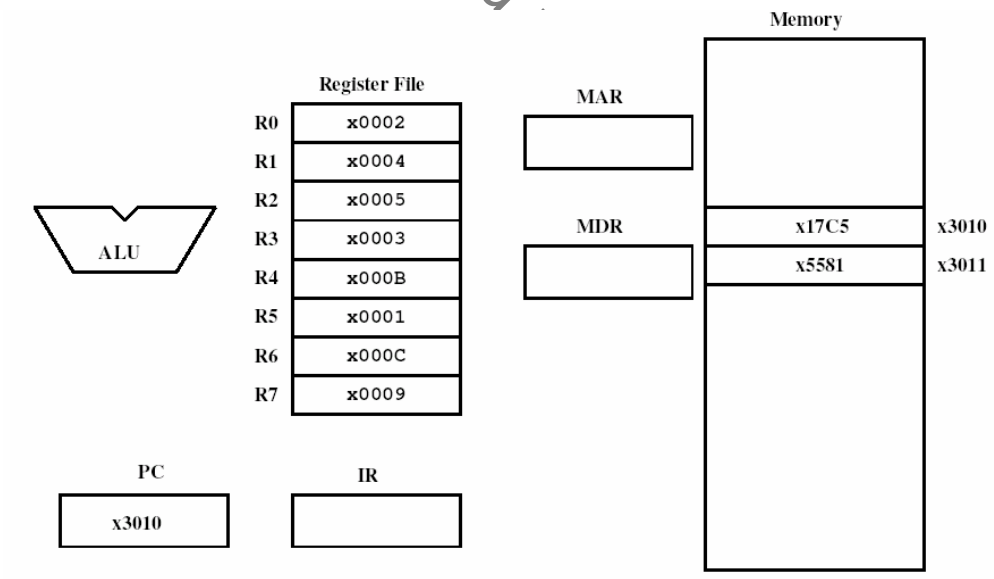
b3) Interpretaţi locaţia 6 ca un număr flotant în două ipostaze: – 1 bit de *semn*, 7 biţi de *exponent* şi 8 biţi de *fracţie*, respectiv – 1 bit de *semn*, 5 biţi de *exponent* şi 10 biţi de *fracţie*.

b4) Interpretaţi locaţia 5 ca un întreg fără semn.

c) În modelul de procesare Von Neumann conţinutul unei locaţii de memorie poate fi şi o instrucţiune. Dacă acest lucru se întâmplă la locaţia 0, identificaţi instrucţiunea.

d) O valoare binară poate fi interpretată şi ca o adresă de memorie. Presupunând acest lucru la locaţia 5, ce adresă de memorie se indică prin conţinutul acestei locaţii. Ce valoare binară conţine respectiva locaţie?

6. Se consideră că simulatorul LC-3 urmează să efectueze faza **FETCH** aferentă instrucţiunii de la adresa specificată de registrul PC. Cunoscând configuraţia iniţială a componentelor modelului von Neumann (unitatea de procesare – setul de regiştrii generali, unitatea de control – registrul PC şi unitatea de memorare – regiştrii MAR şi MDR şi respectiv conţinutul locaţiilor de memorie) se cere să se determine conţinutul fiecăreia dintre resursele indicate în figură după executarea de două ori a *ciclului instrucţiunii*. **Indicaţie**: Instrucţiunea de la adresa 0x3010 este una de adunare (ADD) iar cea de la adresa 0x3011 este ŞI Logic (AND).



6. LC-3 – ARHITECTURA SETULUI DE INSTRUCȚIUNI. CALEA FLUXULUI DE DATE. ORGANIZAREA MEMORIEI LA LC-3 [Patt03]

6.1. LC-3 – ARHITECTURA SETULUI DE INSTRUCȚIUNI

Arhitectura Setului de Instrucțiuni (**ISA**) – reprezintă interfața dintre software (programele de aplicație / sistem de operare) și hardware-ul care îl execută. ISA specifică modul de organizare a memoriei (*spațiul de adresare*: zonă de date statice și dinamice, de cod, de stivă, zonă rezervată nucleului sistemului de operare, *adresabilitatea* – număr de biți stocați la fiecare locație), setul de regiștri, setul de instrucțiuni, formatul instrucțiunii, tipurile de date utilizate și modurile de adresare (mecanismul prin care calculatorul / procesorul localizează operanzii). Translatarea unui program de nivel înalt (fie acesta *C*, *Fortran*) în ISA-ul aferent calculatorului care va executa respectivul program (uzual IA-32) se realizează prin intermediul compilatorului.

Capitolul de față realizează trecerea de la partea hardware spre cea software a unui sistem de calcul. Astfel, este descrisă arhitectura setului de instrucțiuni aferentă procesorului virtual (simulatorului) LC-3 (*Little Computer* versiunea *3*) [Patt03] și vor fi analizate detaliat organizarea memoriei și calea fluxului de date respectiv control la LC-3.

Într-o altă definiție, ISA ar reprezenta totalitatea componentelor și operațiilor (*hardware* ale) unui calculator vizibile la nivelul programatorului (*software*). ISA pune la dispoziția proiectantului (hardware) de sisteme de calcul toate informațiile necesare pentru a putea construi un calculator conform modelului propus de *von Neumann*. Aceste informații sunt suficiente și pentru cineva care vrea să scrie un program în limbajul mașină al respectivului procesor sau să înțeleagă dacă un program scris în limbaj de nivel înalt a fost translatat corect în codul mașină al procesorului în cauză.

6.1.1. LC-3 ISA: ORGANIZAREA MEMORIEI ŞI SETUL DE REGIŞTII GENERALI

Spaţiul de memorie aferent unui sistem de calcul este organizat în aşa fel încât să reţină **instrucţiunile** programelor utilizator, instrucţiunile programelor supervizor (aferente sistemului de operare) dar şi **datele** prelucrate prin intermediul respectivelor programe.

După cum s-a mai precizat în capitolul anterior, arhitectura von Neumann a unui sistem de calcul pune la dispoziţie doi regiştrii de interfaţă procesor – memorie:

- Registrul de adresă al memoriei (MAR) care selectează prin intermediul unui decodificator de adresă locaţia de memorie care va fi scrisă sau citită.
- Registrul de date al memoriei (MDR) care reţine data citită sau care se va scrie din / în memorie.

LC-3 reprezintă un procesor virtual pe 16 biţi (fiecare instrucţiune este codificată pe 16 biţi). De asemenea, memoria este adresată printr-un cuvânt[5] de 16 biţi (busul de date fiind de 16 biţi). În consecinţă, spaţiul de memorie adresabil la LC-3 este de 2^{16} locaţii = 65536 (64k) locaţii. Adresabilitatea la LC-3 este tot de 16 biţi, de unde rezultă faptul că, LC-3 dispune de o memorie totală de 64k × 2o = 128ko. Spre deosebire de alte procesoare, LC-3 nu este adresabilă pe octet. Spaţiul de memorie al LC-3 (pentru detalii a se vedea şi figura 10.4 din capitolul 10) conţine o zonă de cod utilizator (începând cu adresa 0x3000), zona de date globale, zona rezervată sistemului de operare (care include şi instrucţiunile rutinelor de serviciu), zona de stivă (aferentă apelurilor de funcţii din programele utilizator – variabile locale, parametrii, etc.).

Întrucât şi din punct de vedere logic dar şi fizic memoria diferă de (este situată în afara) unitatea de procesare, operaţiile cu memoria consumă de cele mai multe ori mai mult decât un ciclu de tact procesor ($>1T_{CPU}$). Aceste operaţii presupun întâi calculul adresei şi apoi citirea sau scrierea datei, realizabile prin instrucţiuni *load – LDR* sau *store – STR*.

Setul de regiştrii generali alături de unitatea aritmetico-logică compun unitatea de procesare a oricărui sistem de calcul. Regiştrii procesorului constituie o resursă de memorare temporară de viteză foarte mare (accesul – scrierea şi citirea – se face la viteza procesorului) dar de

[5] Cu toate că, la toate procesoarele octetul este format din 8 biţi, numărul de biţi care compun un cuvânt de date nu este identic la toate arhitecturile. Astfel, la LC-3 cuvântul (*word*) este pe 16 biţi iar la procesorul MIPS R3000 este pe 32 de biţi, informaţia stocată pe 16 biţi numindu-se semi-cuvânt (*halfword*).

capacitate redusă. Pot fi accesați pe timpul fazei de procesare *Fetch Operand* aferentă instrucțiunilor de tip Add, Load sau Store. LC-3 [Patt03] dispune de 8 regiştrii generali $R_0 \div R_7$, fiecare pe 16 biți şi trei regiştrii booleeni de condiție (N – negativ, Z – zero, P – pozitiv) fiecare pe 1 bit, setați sau resetați de către instrucțiunile care au un registru destinație (aritmetico-logice şi de citire din memorie). În fiecare moment, cel mult un registru de condiție este setat, bazat pe ultima instrucțiune care alterează un registru general. Un alt registru foarte important, indirect adresabil de către programator este PC (*Program Counter*), care reține adresa următoarei instrucțiuni. Prin instrucțiunile de salt şi apel de / revenire din subrutină valoarea PC-ului poate fi alterată.

6.1.2. LC-3 ISA: FORMATUL INSTRUCȚIUNII ŞI SETUL DE INSTRUCȚIUNI

Se reamintește că unitatea de control a procesorului LC-3 [Patt03] dispune de 2 regiştrii: PC şi IR (registrul care reține instrucțiunea aflată în curs de procesare). Instrucțiunile procesorului LC-3 sunt codificate pe un singur cuvânt[6] de 16 biți şi sunt compuse din două părți principale:

- ***Opcode*** (sau codul operației, reținut de cei mai semnificativi 4 biți ai registrului instrucțiunii – **IR[15:11]**). Rezultă practic 2^4=16 operații (instrucțiuni) distincte – un set foarte simplu. În general trebuie ales un compromis între un set de instrucțiuni complex, dar de cele mai multe ori redundant, caracterizat de un cost ridicat şi un set optimizat de instrucțiuni la un cost mai mic (vezi clasificarea procesoarelor RISC – CISC [Vin03] care face obiectul cursului de *Organizarea şi proiectarea microarhitecturilor*). Acest compromis se stabileşte de cele mai multe ori pe baza simulării pe programe de test reprezentative (*benchmark-uri*).
- Cei mai puțini semnificativi 12 biți ai registrului instrucțiunii (**IR[11:0]**) specifică **operanzii** (asupra cărora se aplică operația) conform **modului de adresare** aferent instrucțiunii:
  - **Regiştrii**: câte 3 biți pentru fiecare operand – sursă şi / sau destinație.
  - **Câmp generator de adresă** (*offset*) – pe 6, 9 sau 11 biți.
  - **Valoare imediată –** pe 5 biți.

[6] Există procesoare (de exemplu INTEL) cu instrucțiuni de lungime variabilă – pe unul sau mai multe cuvinte.

Singurul **tip de dată nativ** (implementat la) **LC-3** este tipul **întreg complement faţă de 2** pe 16 biţi. Celelalte procesoare (de exemplu MIPS, INTEL) implementează şi tipurile: întreg pe 8 biţi cu şi fără semn (*short / byte, byte / unsigned byte*), întreg pe 16 biţi cu şi fără semn (*int / unsigned int*, *halfword* / *unsigned halfword*), întreg pe 32 biţi cu şi fără semn (*long / unsigned long*, *word* / *unsigned word*), simplă precizie virgulă mobilă – pe 32 de biţi (*float*) respectiv dublă precizie virgulă mobilă – pe 64 de biţi (*double*).

Modurile de adresare specifică modul de localizare al operanzilor instrucţiunii. Prin convenţie, **adresa efectivă (EA)** reprezintă locaţia de memorie a operandului. LC-3 [Patt03] suportă 5 moduri de adresare:

- **Imediat** (dacă operandul este localizat direct în instrucţiune).
- **Registru** (operanzii sursă sunt doi regiştrii generali).
- **Memorie** (cu:
  - **Adresare directă** (sau relativă la PC): EA este codificată în corpul instrucţiunii şi se obţine printr-o însumare a PC cu câmpul generator de adresă pe 9 biţi).
  - **Adresare indirectă**. În corpul instrucţiunii este codificat un pointer[7] spre EA: valoarea PC + câmp generator de adresă pe 9 biţi formează adresa la care găsim adresa operandului (şi nu operandul ca în cazul adresării directe).
  - **Adresare indexată** (sau relativă la o adresă de bază) – folosită în cazul instrucţiunilor de prelucrare ale tablourilor (structurilor matriceale). EA se obţine însumând adresa de bază stocată într-un registru general şi un câmp generator de adresă pe 6 biţi. Diferenţa dintre adresarea directă şi cea indexată constă în faptul că, în cazul primeia, adresarea se face într-o zonă (NEAR) – apropiată PC-ului instrucţiunii de transfer date (+/- 256 instrucţiuni) iar în cazul celei de-a doua adresări, spaţiul accesat poate fi (FAR) îndepărtat faţă de instrucţiunea de transfer (practic aproape oriunde în zona de cod utilizator).

Pe scurt, setul de instrucţiuni aferent LC-3 ISA cuprinde:

- **Instrucţiuni operaţionale**, care manipulează direct date. Din această categorie fac parte cele aritmetico-logice (ADD, AND şi NOT).
- **Instrucţiuni de transfer date**:
  a) între memorie şi regiştrii procesorului
     - citire din memorie (LD, LDI şi LDR)

[7] Termenul de *pointer* a fost preluat în limba română şi poate fi folosit cu sensul de referinţă, indicator de adresă, localizator. Pentru detalii studiaţi capitolul 12.

- scriere în memorie (ST, STI şi STR)
- încărcarea unei adrese de bază într-un registru destinație (LEA). Spre deosebire de cele 6 instrucțiuni anterioare, LEA nu accesează memoria.

b) între memorie / regiştrii şi *porturi* (regiştrii de interfață ai dispozitivelor periferice – regiştrii de stare sau de date ai tastaturii sau ai monitorului – vezi pentru detalii capitolul 8 *Întreruperi software*).

❖ **Instrucțiuni de control** (salt condiționat / neocndiționat, direct / indirect şi respectiv apel / revenire de / din subrutină, întreruperi software): BR, JMP / RET, JSR / JSRR, TRAP, RTI. Efectul tuturor acestor instrucțiuni este de a modifica cursul programului (fluxul de execuție). Practic, indirect se modifică PC-ul (adresa următoarei instrucțiuni de executat din program).

6.1.2.1. LC-3 ISA: INSTRUCȚIUNI OPERAȚIONALE

LC-3 implementează doar trei instrucțiuni aritmetico-logice: ADD, AND şi NOT. Operanzii destinație ai acestora sunt regiştrii generali ai procesorului dar semnul acestuia va seta şi unul din cei trei regiştrii booleeni (N, Z sau P) resetându-i pe ceilalți doi. Aceste instrucțiuni nu referă zone de memorie. La LC-3 ISA nu există posibilitatea adunării conținutului unei locații de memorie cu o valoare imediată şi stocării rezultatului înapoi în memorie prin intermediul unei singure instrucțiuni ca la IA-32 ISA. Acest lucru poate fi realizat printr-o secvență de trei instrucțiuni: una de citire din memorie, una de adunare şi una de scriere înapoi în memorie.

Instrucțiunile ADD şi AND pot opera şi în modul imediat de adresare. Pentru fiecare din instrucțiunile setului LC-3 ISA sunt ilustrate grafic codificarea şi succesiunea de operații efectuate (vezi figurile 6.1 ÷ 6.14) pentru execuția cu succes a instrucțiunilor.

LC-3 nu implementează (încă! – poate la versiunile viitoare) instrucțiunile aritmetice mari consumatoare de timp – înmulțirea şi împărțirea, ci doar instrucțiunea de adunare (deoarece se operează cu numere întregi în Complement față de 2 nu este necesară implementarea scăderii, aceasta fiind de fapt o adunare cu inversul operandului). Întrucât s-a arătat în capitolul 3 (vezi şi [Patt03]) „*completitudinea porților ŞI-NU*" – adică prin intermediul acestor porți pot fi implementate oricare operații logice – LC-3 ISA oferă doar instrucțiunile NOT şi AND, celelalte instrucțiuni (OR, XOR) existente la alte arhitecturi (MIPS, INTEL) putând

fi realizate prin combinaţii ale instrucţiunilor NOT şi AND (vezi spre rezolvare problemele 9 şi 10 de la sfârşitul acestui capitol).

În continuare, prin **Reg** identificăm setul de regiştrii generali ai procesorului, câmpul **Dst** substituie codul (iar Reg[Dst] numele) registrului destinaţie iar câmpul **Src** substituie codul (iar Reg[Src] numele) registrului sursă. De asemenea, **BaseR**, folosit în cazul instrucţiunilor de transfer cu mod de adresare indexat codifică registrul de bază (adresa de început de pagină sau zona de memorie utilizator supusă atenţiei – începutul unui tablou unidimensional), de la care se va face un acces relativ în acea pagină.

Instrucţiunea NOT

NOT Reg[Dst], Reg[Src]

NOT reprezintă un operator unar (are un singur operand sursă). După cum se poate vedea şi din figura 6.1, registrul destinaţie este dat de câmpul IR[11:9] iar registrul sursă de câmpul IR[8:6], restul biţilor fiind 1 (IR[5:0]).

Semantica instrucţiunii (Reg[Dst] <= NOT Reg[Src]) presupune inversarea fiecărui bit al registrului sursă (transformarea în complement faţă de 1) şi copierea pe poziţia corespunzătoare în registrul destinaţie. Registrul sursă rămâne nemodificat în urma execuţiei instrucţiunii. De menţionat că sursa şi destinaţia pot referi acelaşi registru.

Exemplu:

NOT R2, R6

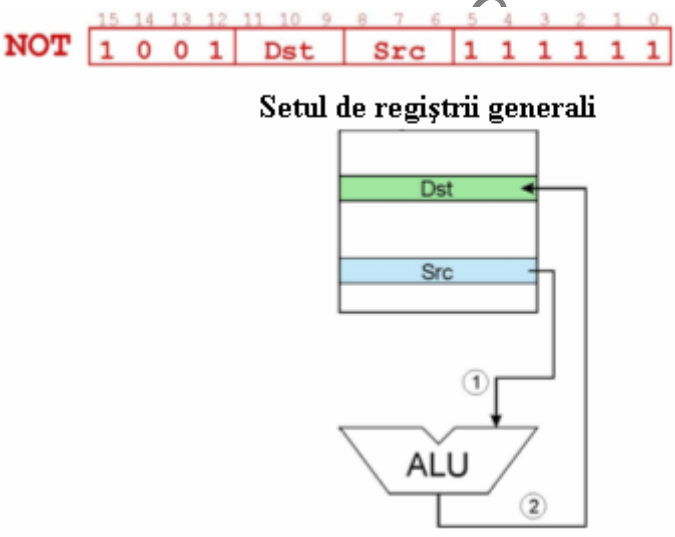


Figura 6.1. Codificarea instrucţiunii NOT şi succesiunea de operaţii efectuate în cazul acesteia

La începutul fazei de *decodificare* câmpul IR[8:6] va ataca setul de regiştrii generali de unde, pe timpul fazei *fetch operand*, registrul sursă *Src* va ataca o intrare a unității aritmetico-logice (ALU). Prin intermediul opcode-ului (IR[15:12]) se atacă simultan automatul cu stări finite (vezi figura 6.17 iar pentru detalii subcapitolul 4.2) – care va genera comanda (semnalul) NOT pentru unitatea ALU. Declanşarea acestuia (NOT) se realizează în faza de *execuție* urmată apoi de scrierea rezultatului în setul de regiştrii generali (pe timpul fazei *scriere rezultat*).

Instrucțiunile ADD/AND

ADD Reg[Dst], Reg[Src1], Reg[Src2] sau **ADD Reg[Dst], Reg[Src1], Imm5**
şi
AND Reg[Dst], Reg[Src1], Reg[Src2] sau **AND Reg[Dst], Reg[Src1], Imm5**

Instrucțiunile ADD/AND au modul de operare similar. Sunt operații binare (au doi operanzi sursă – fie doi regiştrii, fie un registru şi o valoare imediată). Registrul destinație este identificat prin câmpul IR[11:9] iar unul dintre regiştrii sursă prin câmpul IR[8:6]. Dacă modul de adresare este imediat atunci bitul IR[5] = 1 iar câmpul IR[4:0] specifică valoarea imediată, pe 5 biți, căreia i se va extinde semnul pe16 biți obținându-se un număr întreg în complement față de 2. Se reaminteşte că, operațiile aritmetico-logice se efectuează cu operanzi de dimensiuni egale în număr de biți. Semantica celor două instrucțiuni este următoarea:

Reg[Dst] <=[8] Reg[Src1] + SEXT(Imm5) – în cazul adunării şi respectiv

Reg[Dst] <= Reg[Src1] AND SEXT(Imm5) – în cazul operației de ŞI *logic*

De exemplu, inițializarea unui registru cu 0 rezultă în mod natural printr-o instrucțiune care face AND cu valoarea 0. De exemplu, în cazul instrucțiunii AND R2, R2, #0, deşi valoarea 0 este pe 5 biți reprezentată în registrul instrucțiunii, prin extensia semnului (care este tot 0) rezultă 0 pe 16 biți. Simbolul ”#” (a se citi *diez*) semnifică faptul că valoarea ce îi urmează va fi în sistemul zecimal de numerație. O altă posibilitate ar fi fost precedarea valorii de şirul „0x” care sugerează o valoare în sistemul hexazecimal de numerație.

[8] Simbolul <= are rolul de atribuire a valorii expresiei din dreapta sa variabilei din stânga sa.

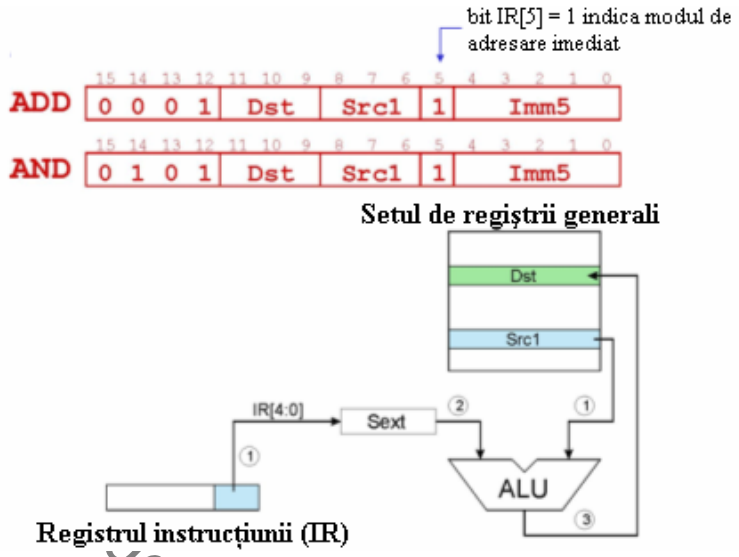


Figura 6.2. Codificarea instrucţiunilor ADD/AND cu mod de *adresare imediat* şi succesiunea de operaţii efectuate în cazul acestora

Dacă modul de adresare este registru atunci bitul IR[5] = 0, biţii IR[4:3] sunt şi ei 0 iar câmpul IR[2:0] specifică cel de-al doilea registru sursă. Semantica celor două instrucţiuni este următoarea:
Reg[Dst] <= Reg[Src1] + Reg[Src2] – în cazul adunării şi respectiv
Reg[Dst] <= Reg[Src1] AND Reg[Src2] – în cazul operaţiei de ŞI *logic*

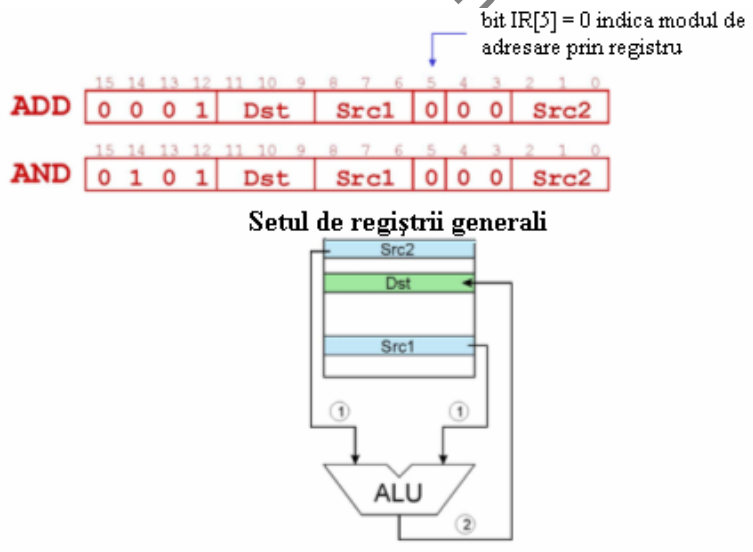


Figura 6.3. Codificarea instrucţiunilor ADD/AND cu mod de *adresare prin registru* şi succesiunea de operaţii efectuate în cazul acestora

Exemple:

```
ADD   R1, R4, R5
AND   R2, R3, R6
```

Figura 6.4 descrie modul de lucru al instrucțiunii: ADD R1, R4, # -2 (decrementare cu 2). Astfel, pe o intrare a unității ALU se va regăsi valoarea din R4 iar pe cealaltă valoarea (-2) reprezentată în complement față de 2. Prin intermediul opcode-ului automatul cu stări finite generează comanda (semnalul) ADD, efectuându-se apoi adunarea şi rezultând valoarea finală (6 – 2 = 4) în R1 (registrul destinație).

De menționat că, LC-3 este implementat ca un procesor cu set optimizat de instrucțiuni (nu implementează instrucțiuni pentru scădere, pentru incrementare / decrementare sau pentru transferul valorilor dintr-un registru într-altul). Totuși, setul redus de regiștrii generali nu permite cablarea unui registru (R_0) la masă, cum fac alte procesoare RISC [Flo03].

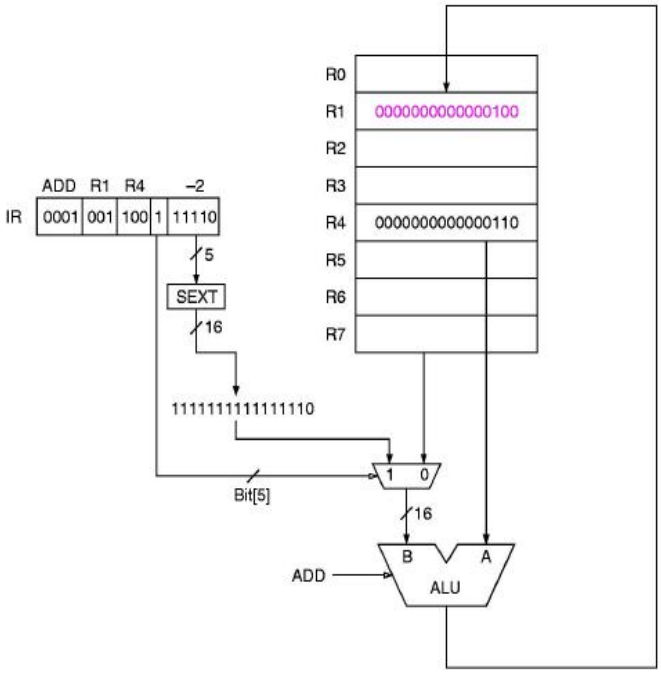


Figura 6.4 Exemplu de adunare cu mod de adresare imediat

6.1.2.2. INSTRUCŢIUNI CU REFERIRE LA MEMORIE (*DE TRANSFER DATE*)

Întrucât LC-3 pune la dispoziţie trei moduri de adresare pentru localizarea datelor în memorie rezultă trei opcode-uri distincte pentru încărcarea datelor din memorie într-un registru: LD (*load direct*), LDI (*load indirect*) şi LDR (*load indexat*) şi respectiv, trei instrucţiuni distincte pentru scrierea datelor din regiştrii procesorului în memorie: ST (*store direct*), STI (*store indirect*) şi STR (*store indexat*).

În linii mari, formatul celor 7 instrucţiuni de transfer date (LD, LDI, LDR, LEA, ST, STI şi STR) este ilustrat în tabelul 6.1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cod operaţie (***opcode***)				**Dst** sau **Src**			Câmp generator de adresă (*pgoffset9*)								

Tabelul 6.1. Formatul general al instrucţiunilor de transfer date (*Load / Store*)

Problema principală în cazul instrucţiunilor cu referire la memorie o reprezintă modul de adresare, care intenţionează specificarea adresei direct în instrucţiune. Dar cum spaţiul de adresare este de 2^{16} locaţii şi instrucţiunile sunt pe 16 biţi, mai mult, 4 biţi stabilesc codul operaţiei iar alţi trei biţi registrul sursă / destinaţie, rezultă că doar maxim 9 biţi din instrucţiune pot fi utilizaţi la calculul adresei. Drept soluţie, se poate considera memoria LC-3 ca şi o colecţie de pagini, fiecare de câte 2^9 (512) cuvinte. Adresa de bază poate fi dată de PC (adresare directă, relativă la PC) sau indexată (relativă la un alt registru), iar cei 9 biţi (sau 6 la adresarea indexată) constituie deplasamentul (*offset-ul*) în respectiva pagină.

Instrucţiunile LD şi ST

LD Reg[Dst], ***Label*** şi **ST Reg[Src],** ***Label***

unde *Label* reprezintă adresa efectivă (EA) fie sub formă numerică fie simbolică. EA se determină însumând la PC-ul instrucţiunii LD câmpul generator de adresă din registrul instrucţiunii (*pgoffset9* – din tabelul 6.1) căruia i se extinde semnul de la 9 la 16 biţi. Astfel,

EA <= (PC) + SEXT(IR[8:0])

Evident că şi procedeul invers este valabil, adică, dându-se valoarea lui Label şi a PC-ului instrucţiunii LD se poate determina conţinutul câmpului IR[8:0].

Semantica celor două instrucţiuni este următoarea:

LD Reg[Dst], *Label*	Reg[Dst] <= Mem[(PC) + SEXT(IR[8:0])]
ST Reg[Src], *Label*	Reg[Src] => Mem[(PC) + SEXT(IR[8:0])]

În cele două mnemonici asamblare LD Reg[Dst], *Label* şi ST Reg[Src], *Label* adresa specificată prin eticheta *Label* trebuie să aibă valoarea în intervalul [PC-256; PC+255]. În plus, în cazul instrucțiunilor cu referire la memorie care modifică valoarea unui registru destinație (LD, LDI, LDR şi LEA) codurile de condiție (N, Z şi P) sunt setate corespunzător.

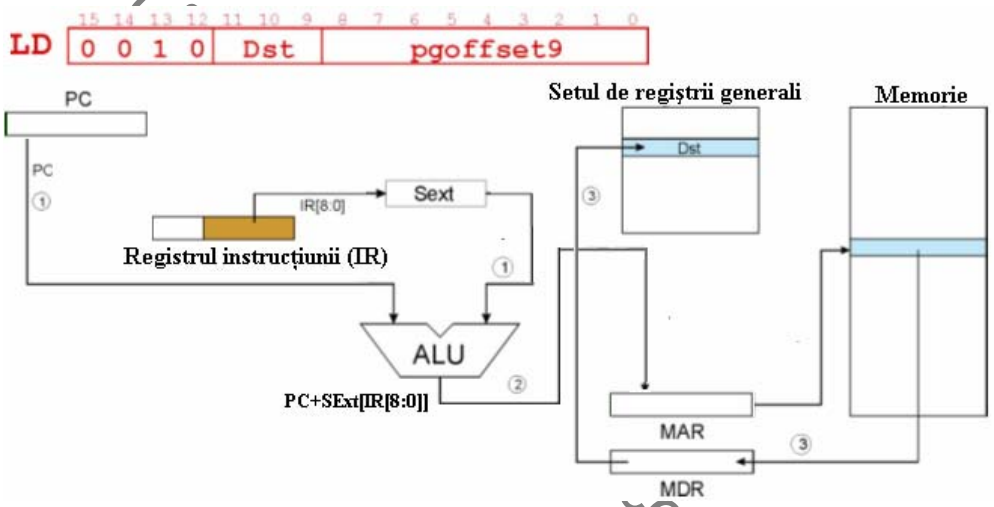


Figura 6.5. Codificarea instrucțiunii Load cu mod de *adresare direct* şi succesiunea de operații efectuate în cazul acestora

În figura 6.5 sunt prezentate codificarea instrucțiunii LD şi succesiunea operațiilor ce concură la execuția acesteia. Astfel, pe timpul fazei de *decodificare*, PC-ul instrucțiunii LD şi câmpul generator de adresă IR[8:0] căruia i se extinde semnul pe 16 biți atacă unitatea ALU (etapa 1 din figura 6.5). Pe timpul fazei de *evaluare adresă* se însumează cele două informații rezultatul depunându-se în registrul MAR (etapa 2 din figura 6.5). În faza de *fetch operand* se accesează memoria cu adresa tocmai stocată în MAR iar conținutul locației de memorie se depune în registrul MDR. În final, pe parcursul fazei *Scrie rezultat*, valoarea din MDR este depusă în setul de regiştrii generali la locația specificată de câmpul Dst (IR[11:9]) (etapa 3).

Exemplu:

Se consideră la adresa PC = 0x4019 instrucţiunea **LD R2, 0x1AF**, iar în memorie la locaţia 0x3FC8 se află valoarea 0x5. Figura 6.6 ilustrează modul de lucru al instrucţiunii LD cu stocarea rezultatului final în registrul destinaţie R2.

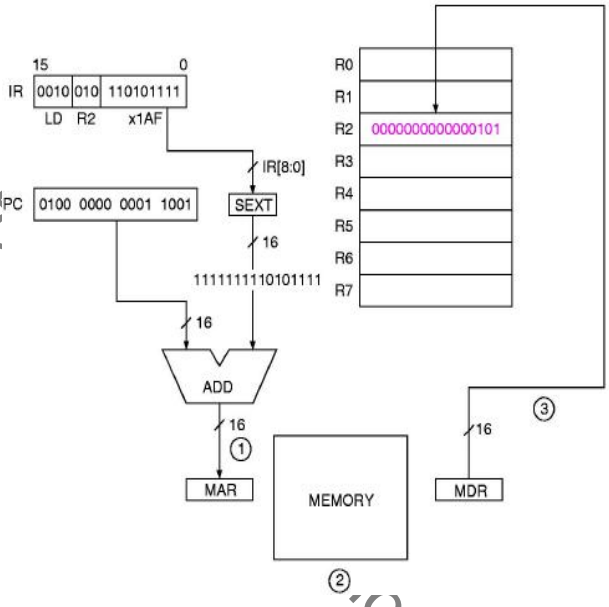


Figura 6.6. Exemplu de instrucţiune Load cu mod de adresare direct

Pentru operaţia de scriere în memorie (instrucţiunea **Store** cu mod de adresare direct) diferenţa în ce priveşte codificarea este următoarea:

0011 Src pgoffset9

În urma execuţiei instrucţiunilor de scriere în memorie (ST, STI şi STR) codurile de condiţie rămân nemodificate. În ce priveşte modul de operare a instrucţiunii Store cu mod de *adresare direct*, succesiunea de operaţii efectuate este aceeaşi cu cea din figura 6.5 cu deosebirea că operaţia 3 se face în sens invers (dinspre setul de regiştrii generali, de la un registru sursă de această dată, spre o locaţie de memorie, prin intermediul acelaşi registru MDR (registru de date a memoriei).

Instrucţiunile LDI şi STI

LDI Reg[Dst], *Label* şi **STI Reg[Src], *Label***

O altă modalitate de a obţine o adresă completă (pe 16 biţi) prin intermediul unei instrucţiuni o reprezintă citirea adresei dintr-o locaţie de

memorie urmată apoi de citirea sau scrierea de la / la respectiva adresă. Instrucțiunile care implementează acest mecanism se numesc cu acces indirect la memorie (*load indirect* sau *store indirect*). În primul rând adresa este obținută din PC-ul instrucțiunii LDI și câmpul generator de adresă (IR[8:0]) căruia i se extinde semnul pe 16 biți (la fel ca la adresarea directă) iar apoi, în al doilea rând, conținutul locației de la adresa calculată este folosit ca și adresă pentru *load* și *store*. Ca și avantaj trebuie spus că această instrucțiune nu necesită un registru de bază pentru adresare. Însă, dezavantajul major îl constituie accesul suplimentar la memorie (două accese pentru obținerea operandului). Instrucțiunile de la nivel *high* (*HLL*) care generează la nivel *low* (asamblare / cod mașină) instrucțiuni cu referire la memorie în mod de adresare indirect sunt cele care operează asupra pointerilor (transmiterea parametrilor unei funcții prin pointer, pointer la pointer).

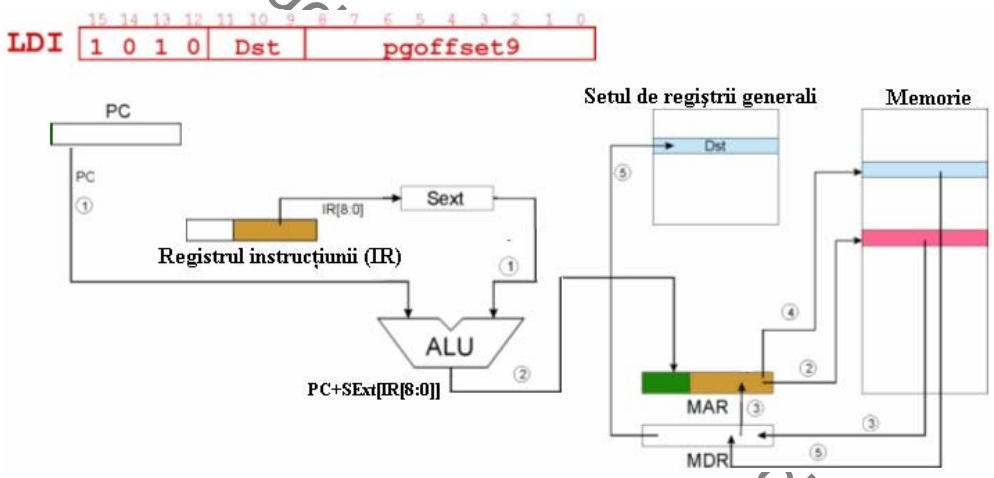


Figura 6.7. Codificarea instrucțiunii Load cu mod de *adresare indirect* și succesiunea de operații efectuate în cazul acestora

În mnemonica asamblare aferentă celor două instrucțiuni *Label* reprezintă adresa adresei („*pointer address*") fie sub formă numerică fie simbolică. Se determină însumând la PC-ul instrucțiunii LDI câmpul generator de adresă din registrul instrucțiunii (*pgoffset9* – din figura 6.7) căruia i se extinde semnul de la 9 la 16 biți. Astfel,

pointer address <= (PC) + SEXT(IR[8:0])

EA (adresa efectivă) <= Mem [*pointer address*]

Reg{Dst] <= Mem [Mem [*pointer address*]]

Evident că şi procedeul invers este valabil, adică, dându-se valoarea lui Label şi a PC-ului instrucţiunii LDI se poate determina conţinutul câmpului IR[8:0].

Modul de operare al instrucţiunii LDI este următorul: pe timpul fazei de *decodificare* se atacă sumatorul (unitatea ALU) cu PC-ul instrucţiunii LDI şi câmpul generator de adresă IR[8:0] căruia i se extinde semnul pe 16 biţi (etapa 1 din figura 6.7). În faza de *evaluare adresă* se obţine adresa adresei operandului, care se depune în registrul MAR (etapa 2 din figura 6.7). Pe parcursul fazei *fetch operand* se atacă memoria cu MAR şi conţinutul locaţiei de memorie se depune în registrul MDR, care ulterior trece în MAR (etapa 3 din figura 6.7). În acest moment în MAR se află adresa operandului. În faza de *execuţie* se accesează memoria din nou cu adresa tocmai stocată în MAR (etapa 4 din figura 6.7) şi se obţine operandul care se depune în registrul MDR (etapa 5 din figura 6.7). În final, pe parcursul fazei *Scrie rezultat*, conţinutul lui MDR este depus în setul de regiştrii generali la locaţia specificată de câmpul Dst (IR[11:9]) (etapa 5).

Exemplu:

Se consideră la adresa PC = 0x4A1C instrucţiunea **LDI R3, 0x1CC**, iar în memorie la locaţia 0x49E8 se află valoarea 0x2110. De asemenea, la locaţia 0x2110 se găseşte valoarea 0xFFFF. Figura 6.8 ilustrează modul de lucru al instrucţiunii LDI cu stocarea rezultatului final în registrul destinaţie R3.

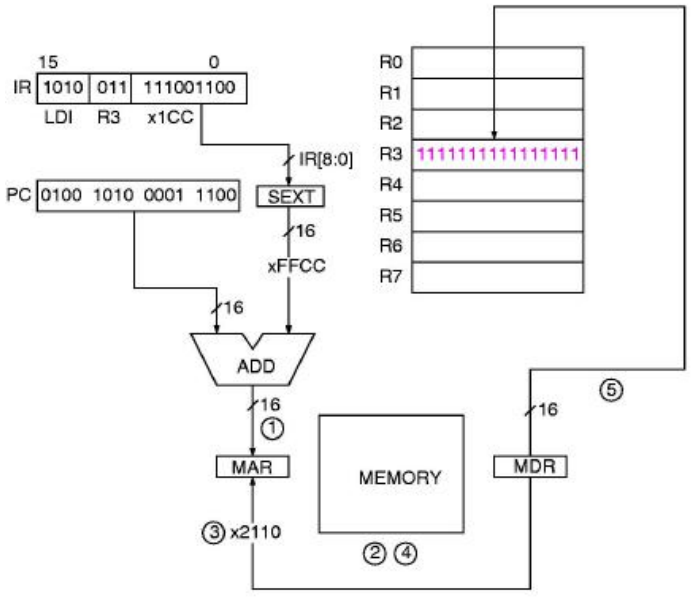


Figura 6.8. Exemplu de instrucţiune Load cu mod de adresare indirect

Pentru instrucțiunea de scriere în memorie în mod de adresare indirect (**STI Reg[Src],** ***Label***) diferența în ce privește codificarea este următoarea:

1011 Src pgoffset9

În ce privește modul de operare a instrucțiunii Store cu mod de *adresare indirect*, succesiunea de operații efectuate este aceeaşi cu cea din figura 6.7 cu deosebirea că, după aflarea adresei efective unde se va scrie informația în memorie, operația 5 se va face în sens invers (dinspre setul de regiştri generali, de la un registru sursă de această dată, spre o locație de memorie, prin intermediul aceluiaşi registru MDR (registru de date a memoriei).

Instrucțiunile LDR şi STR

LDR Reg[Dst], Reg[BaseR], index6 şi **STR Reg[Src], Reg[BaseR], index6**

În modul de adresare direct pot fi adresate cuvinte aflate în aceeaşi pagină de memorie cu instrucțiunea curentă. Totuşi, în cazul programelor mari, acest lucru nu este suficient spațiul fiind prea mic, chiar şi în cazul procesorului LC-3, motiv pentru care proiectanții LC-3 au propus modul de adresare indexat (pentru a accesa zone de memorie din oricare alte pagini, diferite de cea a instrucțiunii curente). Astfel, se foloseşte un registru (pe 16 biți) pentru a furniza adresa de bază. După cum se poate observa, în figura 6.9, codificarea instrucțiunii LDR presupune existența în registrul IR a următoarelor câmpuri: *opcode* pe 4 biți, registrul destinație pe 3 biți, registrul de bază pe 3 biți şi câmpul deplasament (*index6* – din figura 6.9) pe 6 biți (fără semn), deci o valoare întreagă în intervalul [0, 63]. Înainte de a fi adăugat la registrul de bază, în faza de decodificare, se extinde cu 0 câmpul deplasament pe 16 biți. O altă diferență față de modul de adresare direct constă în faptul că, deplasamentul la modul indexat este pe 6 biți, față de 9 biți la cel direct. Adresa efectivă (EA) a operandului din memorie se determină însumând la registrul de bază Reg[BaseR] câmpul generator de adresă din registrul instrucțiunii (*index6*) care se extinde cu 0 de la 6 la 16 biți. Astfel,

EA <= Reg[BaseR] + ZEXT(IR[5:0])

Reg[Dst] <= Mem[Reg[BaseR] + ZEXT(IR[5:0])]

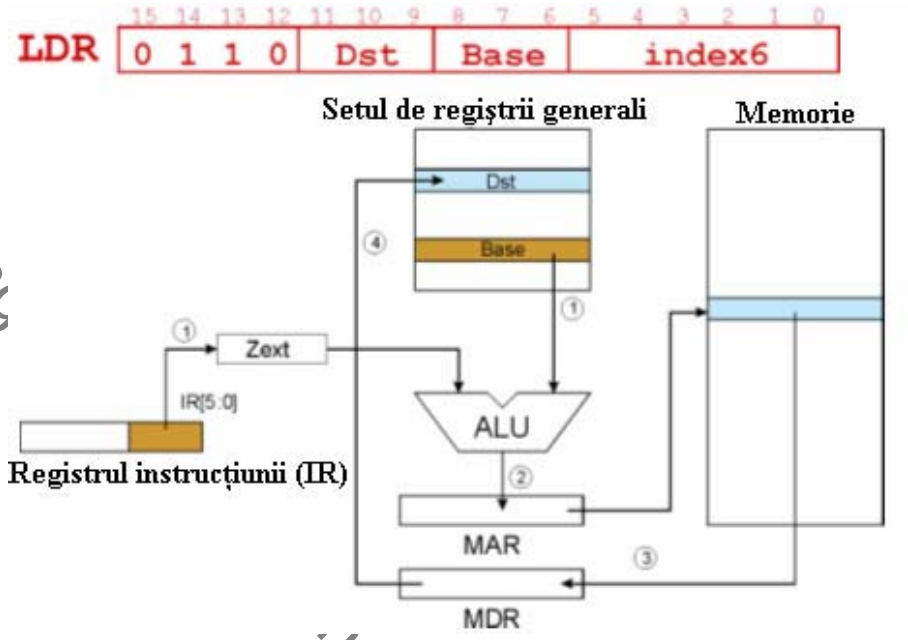


Figura 6.9. Codificarea instrucţiunii Load cu mod de *adresare indexat* şi succesiunea de operaţii efectuate în cazul acestora

Ciclul instrucţiunii LDR (*load indexat*) presupune după aducerea şi depunerea acesteia în registrul IR (lucru efectuat în faza *fetch instrucţiune*) *decodificarea* instrucţiunii. Pe parcursul acestei faze, cu cei trei biţi IR[8:6] se identifică registrul de bază din setul de regiştrii generali şi se realizează extensia cu 0 pe 16 biţi a câmpului deplasament IR[5:0] (etapa 1 din figura 6.9). Pe timpul fazei de *evaluare adresă* se calculează suma dintre cele două informaţii anterior amintite, rezultatul depunându-se în registrul MAR (etapa 2 din figura 6.9). Pe durata fazei de *fetch operand* se accesează memoria de la adresa dată de MAR iar conţinutul se depune în registrul MDR (etapa 3 din figura 6.9), de unde urmează a fi copiat în registrul destinaţie corespunzător în faza *Scrie rezultat* (etapa 4 din figura 6.9).

Exemplu:

Se consideră instrucţiunea **LDR R3, R3, 0x1D**, iar în memorie, la locaţia 0x2362 se află valoarea 0x0F0F. De asemenea, conţinutul registrului R3 înaintea execuţiei instrucţiunii este 0x2345. Figura 6.10 ilustrează modul de lucru al instrucţiunii LDR cu stocarea rezultatului final în registrul destinaţie (acelaşi) R3.

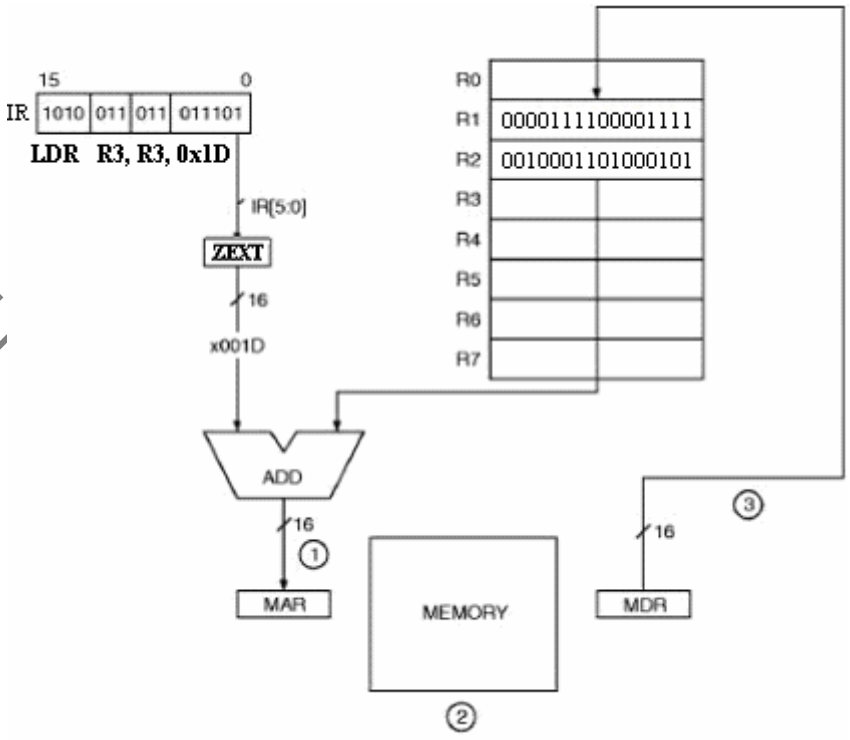


Figura 6.10. Exemplu de instrucțiune Load cu mod de adresare indexat

Pentru instrucțiunea **STR Reg[Src], Reg[BaseR], index6** (scriere în memorie în mod de adresare indexat) diferența în ce privește codificarea este următoarea:

0111 Src BaseR index6

În ce privește modul de operare a instrucțiunii Store cu mod de *adresare indexat*, succesiunea de operații efectuate este aceeași cu cea din figura 6.9 cu deosebirea că, după aflarea adresei efective unde se va scrie informația în memorie, operațiile 3 și 4 se vor face în sens și ordine inversă (4 întâi și 3 după, iar sensul este dinspre setul de regiștrii generali, de la un registru sursă, spre o locație de memorie, prin intermediul registrului MDR.

Instrucțiunea LEA

LEA Reg[Dst], *Label*

unde *Label* reprezintă adresa efectivă (EA) fie sub formă numerică fie simbolică. EA se determină însumând la PC-ul instrucțiunii LEA câmpul

generator de adresă din registrul instrucțiunii (*pgoffset9* – din figura 6.11) căruia i se extinde semnul de la 9 la 16 biți. Astfel,

EA <= (PC) + SEXT(IR[8:0])

Evident că şi procedeul invers este valabil, adică, dându-se valoarea lui Label şi a PC-ului instrucțiunii LEA se poate determina conținutul câmpului IR[8:0].

Semantica instrucțiunii este următoarea şi anume *încarcă în registrul destinație valoarea adresei efective* (nu conținutul de memorie de la adresa).

LEA Reg[Dst], *Label* Reg[Dst] <= (PC) + SEXT(IR[8:0])

Uzual se foloseşte la prelucrarea tablourilor (unidimensionale) situate într-un spațiu de memorie apropiat instrucțiunii curente (+/- 256 de cuvinte) când, se încarcă într-un registru general adresa de început a tabloului, iar accesul la fiecare element al tabloului se face printr-o adresare indexată având ca registru de bază pe cel anterior setat prin instrucțiunea LEA.

Ca şi în cazul adresării directe sau indirecte, adresa specificată prin eticheta *Label* trebuie să fie în intervalul [PC-256; PC+255]. De asemenea, codurile de condiție sunt setate.

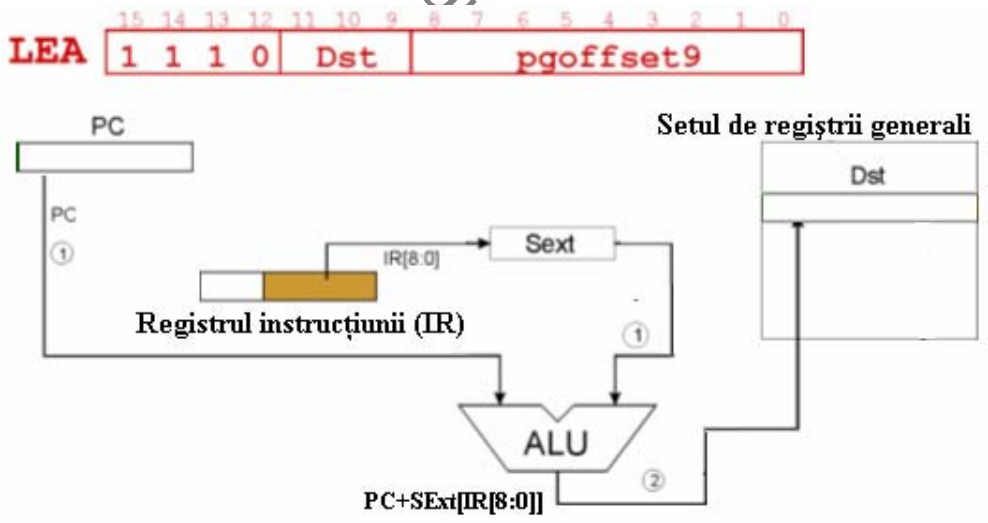


Figura 6.11. Codificarea instrucțiunii Load cu mod de *adresare imediat* şi succesiunea de operații efectuate în cazul acestora

Spre exemplificare, se consideră în memorie la adresa PC=0x4019 următoarea instrucțiune **LEA R5, -3.** Figura 6.12 ilustrează modul de lucru al instrucțiunii LEA cu stocarea rezultatului final în registrul destinație (acelaşi) R5.

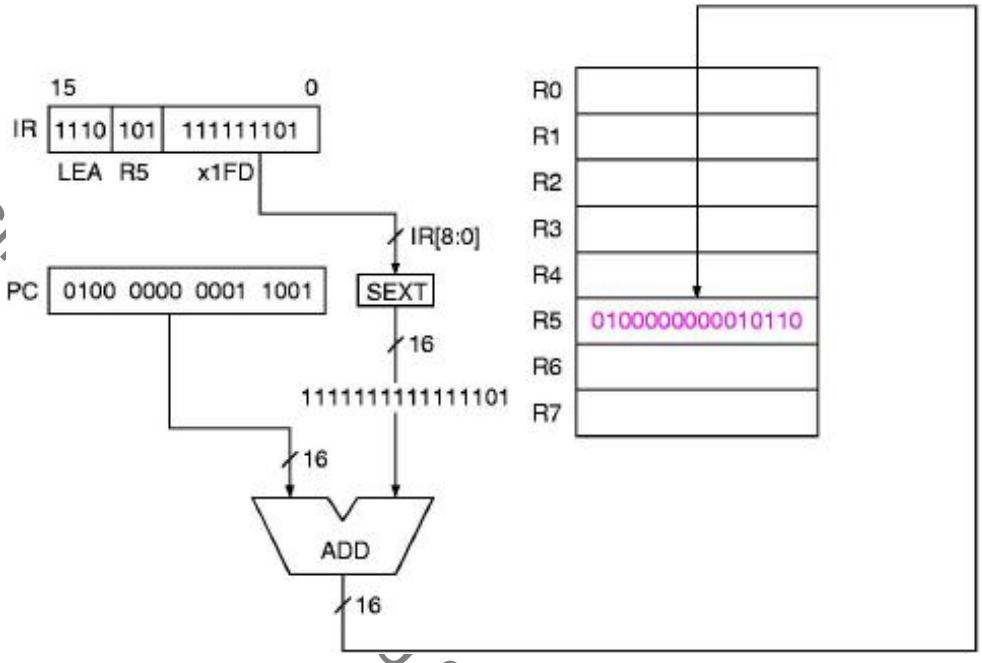


Figura 6.12. Exemplu de instrucțiune Load cu mod de adresare imediat

6.1.2.3. LC-3 ISA: INSTRUCȚIUNI DE CONTROL (*RAMIFICAȚII ÎN PROGRAM*)

Instrucțiunile de ramificație (de salt şi / sau apel) au rolul de a altera cursul secvenței de instrucțiuni (fluxul de control al programului) prin modificarea registrului PC. În funcție de condiția de salt, memorată în corpul instrucțiunii, se cunosc **salturi condiționate** (se face saltul dacă se îndeplineşte condiția) sau **necondiționate** (se face tot timpul saltul – PC-ul se modifică întotdeauna). În cazul saltului condiționat dacă nu se îndeplineşte condiția de salt atunci se va executa următoarea instrucțiune din program (PC-ul obținut în urma fazei *fetch* a instrucțiunii de salt nu este alterat). Salturile necondiționate pot fi **directe** (se cunoaşte tot timpul adresa destinație a saltului – încă din momentul compilării sursei programului) sau **indirecte** (caz în care adresa de salt este dată printr-un registru care-şi modifică în mod dinamic valoarea reținută [Flo05]). În cazul salturilor indirecte, adresa de salt se cunoaşte abia în momentul execuției programului.

Pe lângă instrucțiunile de salt mai există şi instrucțiuni de apel şi revenire din subrutine. Acestea fac parte din categoria ramificațiilor de program necondiționate dar, în care, apelurile au o proprietate importantă:

salvează adresa de revenire în programul apelant (PC-ul instrucțiunii următoare apelului).

Instrucțiunile de întrerupere software – **TRAP** – reprezintă instrucțiuni de apel către rutine de tratare aferente sistemului de operare. La încheierea fiecărei rutine se revine în programul apelant cu o instrucțiune de tip **return** (care repune în registrul PC valoarea adresei succesoare instrucțiunii TRAP).

Pe scurt, instrucțiunile de control oferite de LC-3 ISA sunt [Patt03]:

- **BR x *Label***– salt condiționat (*poate fi totuşi chiar şi necondiționat*) la o adresă relativă la PC-ul instrucțiunii curente de salt (deplasamentul față de PC este pe 9 biți).
- **JMP / JSR *Label*** – instrucțiuni de salt necondiționat (sau apel de subrutină) direct la o adresă pe 11 biți relativă la PC-ul instrucțiunii de salt.
- **JMPR / JSRR** – instrucțiuni de salt necondiționat sau apel de subrutină indirect, la o adresă indexată obținută dintr-un registru de bază şi un deplasament pe 8 biți (fără semn).
- **TRAP** – întreruperi software – practic apeluri directe de subrutine ale sistemului de operare.
- **RET / RTI** – instrucțiuni de revenire din subrutina utilizator sau cea aferentă sistemului de operare.

Instrucțiunea BR x *Label*

unde *Label* reprezintă adresa efectivă (EA) fie sub formă numerică fie simbolică. EA se determină însumând la PC-ul instrucțiunii BR câmpul generator de adresă din registrul instrucțiunii (*pgoffset9* – din figura 6.13) căruia i se extinde semnul de la 9 la 16 biți. Astfel,

$$EA <= (PC) + SEXT(IR[8:0])$$

Evident că şi procedeul invers este valabil, adică, dându-se valoarea lui Label şi a PC-ului instrucțiunii BR se poate determina conținutul câmpului IR[8:0].

Simbolul *x* reprezintă condiția care trebuie îndeplinită şi poate fi *n* (negativ), *z* (egalitate cu 0), *p* (strict pozitiv), *nz* (mai mic sau egal cu 0), *np* (diferit de 0), *zp* (mai mare sau egal cu 0), *nzp* (saltul se face necondiționat pe oricare din condiții – este echivalent instrucțiunii JMP, doar că deplasamentul este pe 9 biți).

Semantica instrucțiunii este următoarea: dacă cei trei biți de condiție (regiştrii booleeni N, Z şi P) sunt 0 (nu sunt setați practic), conform figurii

6.14 niciuna din cele trei porți logice ŞI nu generează 1 logic (toate ieşirile sunt 0) şi rezultă că saltul nu se face (echivalent unui simplu NOP – *no operation*) şi se trece la următoarea instrucțiune din program succesoare branch-ului. Dacă registrul boolean corespunzător condiției este setat (de ex. condiția „*n*" – negativ şi registrul N=1) atunci saltul se va face (se spune că este „*taken*") la adresa specificată prin eticheta *Label*, noul PC nemaifiind (PC+1) ci (PC + SEXT(IR[8:0])). Ca şi în cazul acceselor la memorie cu mod de adresare direct, adresa specificată prin eticheta *Label* (*target-ul* saltului) trebuie să fie în intervalul [PC-256; PC+255], unde PC reprezintă adresa instrucțiunii de salt. Dacă nu coincide însă condiția cu regiştrii booleeni setați (de ex. condiția „*n*" – negativ şi registrul N=0) atunci saltul nu se va face (se spune că este *not taken*), instrucțiunea care se va procesa fiind cea succesoare branch-ului din program (de adresă PC+1 – vezi şi figura 6.13).

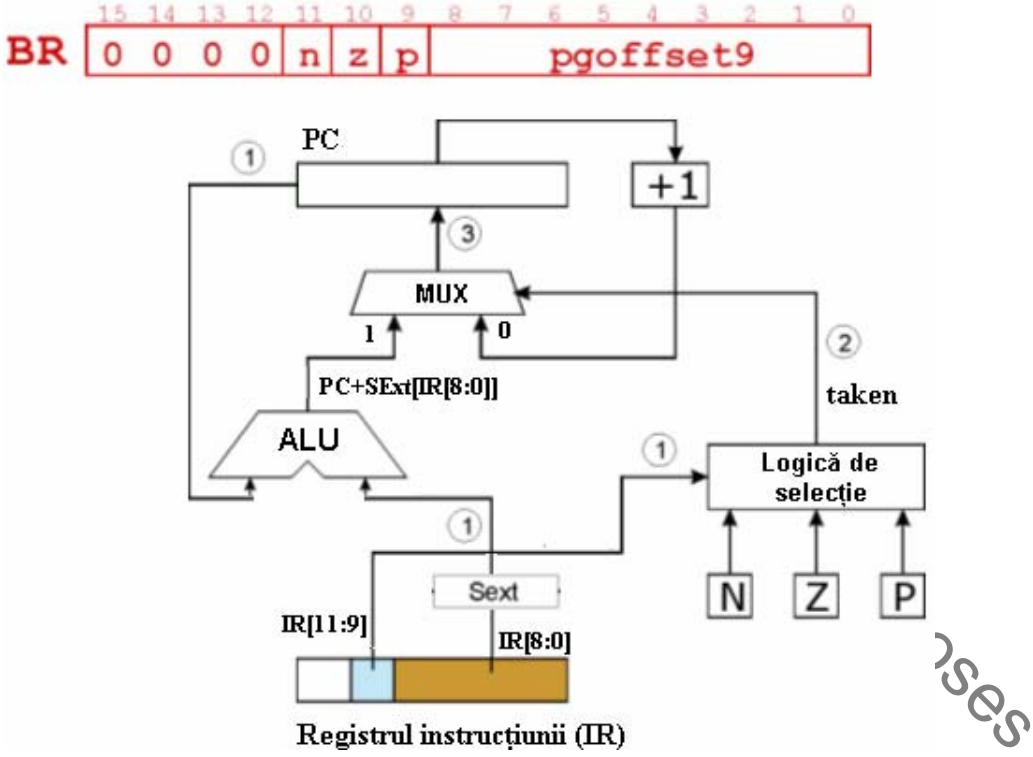


Figura 6.13. Codificarea instrucțiunii *Branch* şi succesiunea de operații efectuate în cazul acesteia

Modul de lucru al instrucțiunii Branch (BR x Label) este următorul: pe timpul fazei *fetch instrucțiune*, practic după depunerea instrucțiunii în

registrul IR are loc şi incrementarea PC-ului cu o unitate pentru a indica spre următoarea instrucţiune din program (valoarea PC+1 – unde PC reprezintă adresa branch-ului – fiind depusă pe intrarea 0 a multiplexorului MUX şi care va fi selectată dacă semnalul *taken* va fi 0, însemnând, de fapt, că saltul nu se face). Pe timpul fazei de *decodificare* câmpul generator de adresă IR[8:0] căruia i se extinde semnul de la 9 la 16 biţi şi PC-ul saltului se depun pe intrarea sumatorului ALU (etapa 1 din figura 6.13) iar câmpul de condiţie IR[11:9] atacă blocul „*logică de selecţie* ” – prezentat detaliat în figura 6.14 (practic 3 porţi logice ŞI cu două intrări şi o poartă logică SAU cu 3 intrări), a cărui ieşire va fi semnalul *taken* (îndeplinirea condiţiei însemnând *se face saltul, altfel nu se face*) din etapa 2 aferentă figurii 6.14. Pe timpul fazei *evaluare adresă* se calculează PC-ul target al branch-ului (PC + SEXT(IR[8:0]) care va ataca a doua intrare a multiplexorului (cea de 1 logic). În faza de *execuţie* în funcţie de îndeplinirea sau nu a condiţiei de salt în PC se încarcă valoarea corectă, adresa instrucţiunii de la care se va procesa în continuare (etapa 3 din figura 6.13).

Spre exemplificare, se consideră instrucţiunea de salt **BR z 0x0D9** stocată în memorie la adresa PC=0x4028. De asemenea, dintre regiştrii booleeni de condiţie doar registrul Z este setat. Figura 6.14 ilustrează succesiunea de operaţii care conduc la modificarea fluxului de control al programului şi determinarea noului PC.

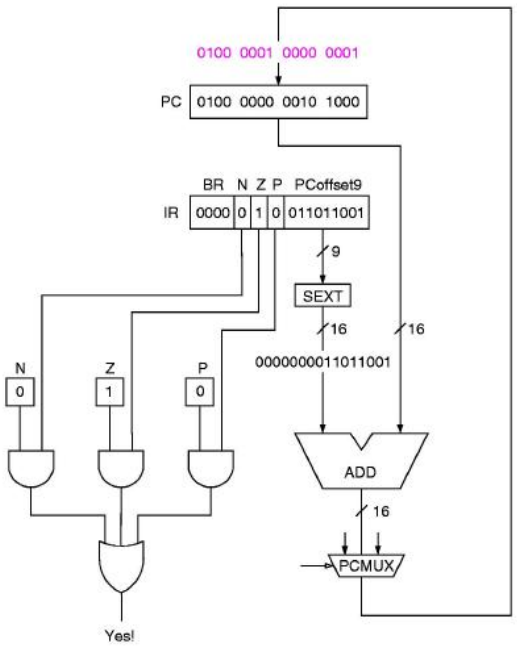


Figura 6.14. Exemplu de instrucţiune de salt condiţionat (*Branch if zero*)

Tratarea instrucțiunilor de salt necondiționat, apel şi revenire din subrutine utilizator şi ale sistemului de operare, precum şi a întreruperilor software *Trap* se va face mai pe larg în capitolul 8 al prezentei cărți.

Instrucțiunile de salt necondiționat (JMP/JSR, JMPR/JSRR la LC-3) se întâlnesc în literatura de specialitate sub numele de „*jump*-uri". Formatul instrucțiunilor JMP şi JSR respectiv JMPR / JSRR este aproape identic, un singur bit diferențiind între salt şi respectiv apel (păstrarea PC-ului de revenire în registrul R7 la LC-3 [Patt03]). Asupra formatului celor patru instrucțiuni (JMP/JSR şi JMPR / JSRR) se va insista în capitolul 8.

Instrucțiunile TRAP sau întreruperile software invocă de fapt o rutină a sistemului de operare identificată printr-un vector de întrerupere pe 8 biți. Cele mai uzuale sunt cele care permit citirea unui caracter de la tastatură, afişarea unui caracter pe ecranul monitorului sau încheierea programului utilizator şi cedarea controlului sistemului de operare. După execuția instrucțiunilor din rutina de tratare a întreruperii software, PC-ul este setat cu adresa instrucțiunii succesoare Trap-ului. Asupra formatului instrucțiunii TRAP, semanticii acesteia, despre adresele de start aferente rutinelor de tratare ale întreruperii se va insista în capitolul 8.

6.1.3. LC-3 ISA: APLICAȚII REZOLVATE

Odată cu descrierea instrucțiunilor de control se cunosc toate tipurile de instrucțiuni pentru realizarea de programe[9]. La nivelul programelor în limbaj de asamblare implementarea buclelor de program poate fi realizată fie prin folosirea unui contor care va fi decrementat la fiecare pas oprirea făcându-se când contorul ajunge la 0 (*structură repetitivă cu număr cunoscut de paşi*) [Neg97, Sto98], fie prin utilizarea unei santinele[10] pe post de control iar oprirea algoritmului se face dacă unul din elementele supuse atenției (citit din memorie) este chiar cel căutat, aflat pe post de santinelă (*structură repetitivă cu număr necunoscut de paşi*).

[9] Teorema Böhm-Jacoppini [Neg97] afirmă că orice program de calcul poate fi realizat folosind structuri secvențiale, alternative şi repetitive. Instrucțiunile de control (de ramificație) sunt elemente de bază ale structurilor alternative şi repetitive. Instrucțiunile operaționale şi de transfer se regăsesc în structurile secvențiale de program.

[10] Un caracter special folosit pentru a indica sfârşitul unei secvențe este numit deseori **santinelă**. Utilitatea sa este foarte mare atunci când nu se cunoaşte apriori numărul de paşi (iterații) pe care îi execută o anumită structură repetitivă.

Aplicaţie 1: Să se calculeze suma a 12 numere întregi. Numerele sunt stocate în memorie începând cu adresa 0x3100. Programul începe la adresa 0x3000. Pentru simplitate, se indică folosirea regiştrilor R1, R2, R3 şi R4, fiecare având următorul rol: R1 – adresa zonei de date (unde sunt stocate numerele), R2 – contorul de numere (indică în fiecare moment câte numere mai sunt de însumat), R3 – suma, R4 – elementul curent citit din memorie.

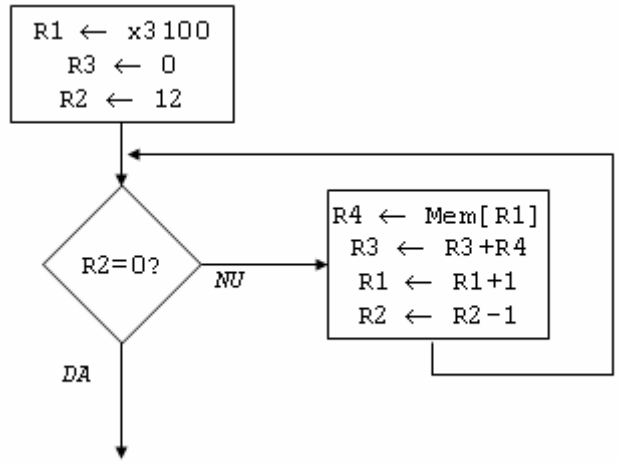


Figura 6.15. Schema logică aferentă programului de însumare a 12 numere întregi

Secvenţa de cod aferentă este următoarea:

PC	**Codificare instrucţiuni**	**Semnificaţie**
0x3000	1110 001 1 0000 0000	*R1 ← x3100*
0x3001	0101 011 011 1 00000	*R3 ← 0*
0x3002	0101 010 010 1 00000	*R2 ← 0*
0x3003	0001 010 010 1 01100	*R2 ← 12*
0x3004	0000 010 000000110	*Dacă Z=1, sare la 0x300A*
0x3005	0110 100 001 000000	*Încarcă următoarea valoare în R4*
0x3006	0001 011 011 0 00 001	*Însumează valoarea citită la suma din R3*
0x3007	0001 001 001 1 00001	*Incrementează adresa de memorie (indicată de R1)*

0x3008	0001 010 010 1 11111	*Decrementează contorul de numere care mai sunt de citit din memorie (R2).*
0x3009	0000 111 111111011	*Sare la testul contorului (adresa 0x3004)*

Aplicația 2: Să se determine numărul de apariții ale unui caracter într-un „*fișier*". Programul este stocat în memorie începând cu adresa 0x3000. caracterul căutat va fi citit de la tastatură. „*Fişierul*" se consideră încărcat de pe discul hard în memorie sub forma unor locații contigue. În fiecare locație se află codul ASCII al unui caracter din fişier. Adresa de început a fişierului este stocată în prima locație de memorie imediat după codul sursă al programului. Dacă ceea ce s-a citit de la tastatură (codul ASCII al caracterului) coincide cu unul din fişier (cu cel citit la momentul respectiv din fişier) se incrementează un contor. Sfârşitul de fişier este indicat prin codul ASCII special **EOT (x04) – "***end of text***".** În final, programul va afişa numărul de caractere din fişier identice cu cel citit de la tastatură. Pentru buna funcționalitate a codului sursă propus trebuie ca numărul de apariții să fie strict mai mic decât 10.

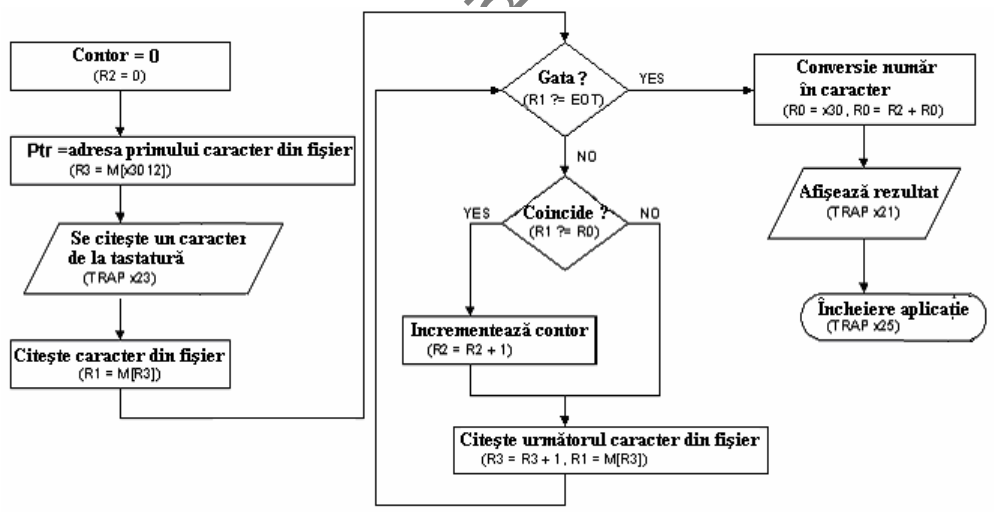


Figura 6.16. Schema logică aferentă programului care determină numărul de apariții ale unui caracter într-un fişier

În secvența de program următoare Contorul (numărul de apariții ale caracterului căutat) este salvat în registrul R2, registrul R3 păstrează adresa de început a fişierului (adresa de bază de la care se va citi pe rând tot câte un caracter). Registrul R1 reține codul ASCII al fiecărui caracter citit din fişier.

Inițial R0 reține codul ASCII al caracterului căutat (cel citit de la tastatură) iar în final codul ASCII al cifrei care reprezintă numărul de apariții al caracterului căutat în fişier.

PC	**Codificare instrucțiuni**	**Semnificație**
0x3000	0101 010 010 1 00000	*R2 ← 0 (contor)*
0x3001	0010 011 000010001	*R3 ← M[x3012] (Ptr)*
0x3002	1111 0000 0010 0011	*Codul caracterului citit de la tastatură se află în R0 (TRAP x23)*
0x3003	0110 001 011 000000	*R1 ← M[R3]*
0x3004	0001 100 001 1 11100	*R4 ← R1 – 4 (EOT)*
0x3005	0000 010 000001001	*Dacă Z=1, sare la adresa 0x300E*
0x3006	1001 001 001 1 11111	*R1 ← NOT R1*
0x3007	0001 001 001 1 00001	*R1 ← R1 + 1*
0x3008	0001 001 001 0 00 000	*R1 ← R1 + R0*
0x3009	0000 101 000000010	*Dacă N=1 sau P=1, sare la adresa 0x300B*
0x300A	0001 010 010 1 00001	*R2 ← R2 + 1*
0x300B	0001 011 011 1 00001	*R3 ← R3 + 1*
0x300C	0110 001 011 000000	*R1 ← M[R3]*
0x300D	0000 111 111110111	*Sare necondiționat la adresa 0x3004*
0x300E	0010 000 000000101	*R0 ← M[x3013]*
0x300F	0001 000 000 0 00 010	*R0 ← R0 + R2*
0x3010	1111 0000 0010 0001	*Afişează numărul de apariţii sub formă de character al cărui cod ASCII se găseşte în R0 (TRAP x21)*
0x3011	1111 0000 0010 0101	*HALT (TRAP x25) – încheie programul*

0x3012	Adresa de început a fişierului	
0x3013	0000 0000 0011 0000	*Codul ASCII al caracterului '0' (0x30)*

6.2. FLUXUL DE DATE LA LC-3 ISA

Foarte sumar, calea (fluxul) de date constă din toate structurile logice care sunt combinate pentru prelucrarea informației în nucleul unui calculator (la nivelul procesorului). Fluxul de date aferent unui sistem de calcul urmează o strategie (un curs) definită la nivelul algoritmului (*software*) şi este susținută printr-o serie de componente *hardware*. Strategia fluxului de date presupune luarea fiecărui element component, determinarea modului de funcționare, identificarea intrărilor / ieşirilor, identificarea logicii de control, a cursului datelor între elementele componente, proiectarea elementelor componente.

Figura 6.17 ilustrează fluxul de date la LC-3 ISA. În figură se disting două tipuri de săgeți: „**pline**" – care indică informația care se procesează (de cele mai multe ori pe 16 biți, dar şi pe 1, 2, 3, 9 sau 11 biți) şi „**goale**" – care indică semnalele de control, generate în majoritatea cazurilor de automatul cu stări finite (*Finite State Machine – FSM*).

Componenta de bază a fluxului datelor la LC-3 o constituie **magistrala globală de date** (*bus*) reprezentată în figura 6.17 cu linia cea mai îngroşată, cu săgeți *pline* la ambele capete. Magistrala globală la LC-3 constă din 16 fire (semnale) şi circuitele electronice asociate. Aceasta permite unei structuri logice să transfere până la 16 biți de informație altei structuri prin conectarea componentelor electronice corespunzătoare la magistrală. În fiecare moment de timp există o singură sursă pe magistrală astfel încât să se transfere o singură valoare. În figura 6.17 se observă că fiecare structură care generează o valoare pe *bus* o face prin intermediul unui dispozitiv ***tri-state*** (sub forma unui triunghi gol cu o latură tăiată) cu rolul de a controla scrierea informației pe magistrală de către o singură sursă (şi nu de mai multe). Structura care doreşte să scrie pe *bus* va avea activat semnalul WE (*write enable*) prin intermediul *unității de control* (mai precis automatul *FSM*). Citirea datelor de pe magistrală poate fi făcută de oricâte structuri logice. Regiştrii capturează data de pe bus doar dacă *unitatea de control* activează semnalul Write Enable pentru destinația corespunzătoare.

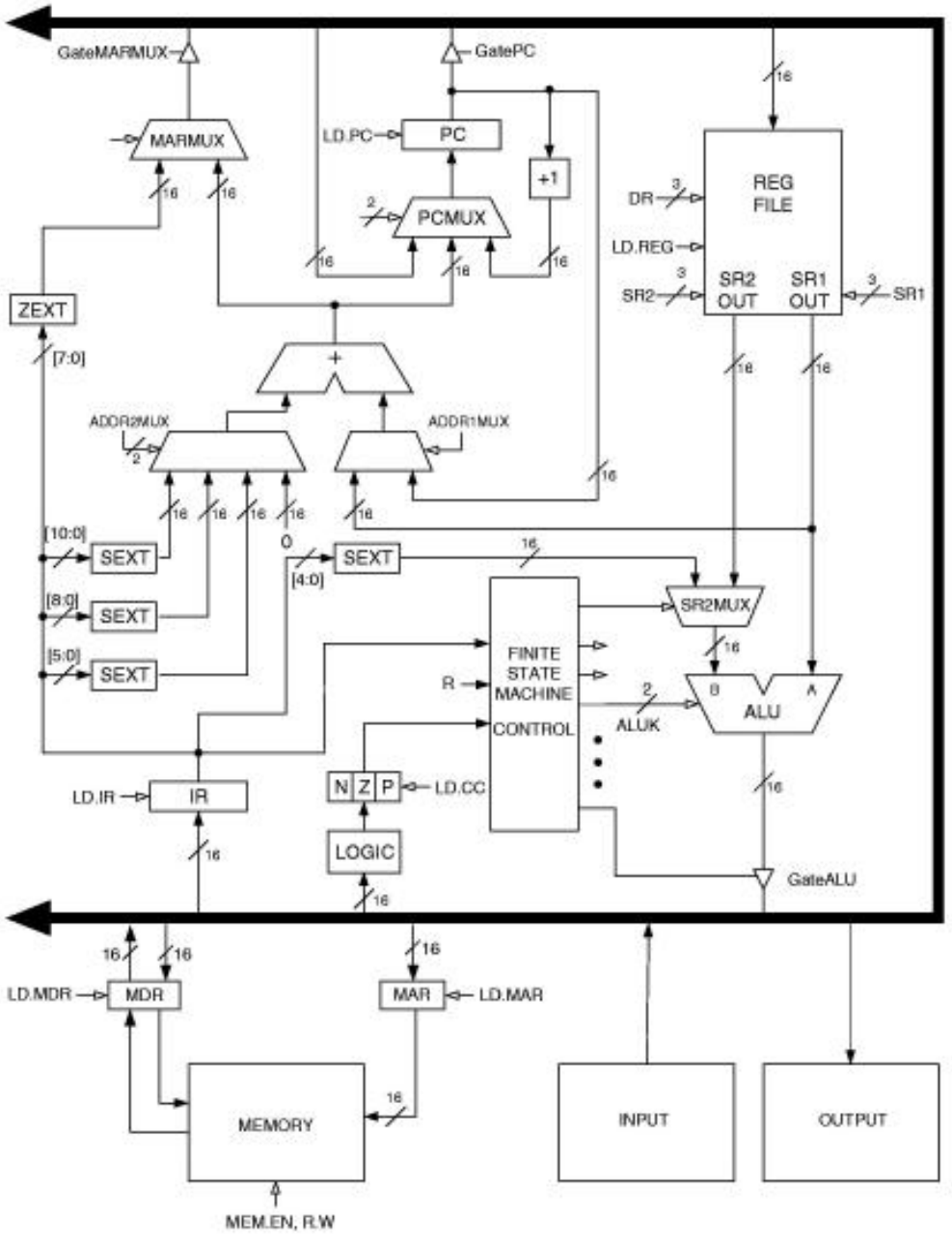


Figura 6.17. Fluxul de date la LC-3 ISA

O componentă, de asemenea, importantă a sistemului de calcul o reprezintă **memoria** care reţine atât instrucţiuni cât şi date. Memoria este accesată prin încărcarea în registrul MAR (registrul de adresă al memoriei) a adresei locaţiei dorite. Apoi, unitatea de control „*activează semnalul READ*”

– la LC-3 WE devine 0, iar după scurgerea timpului de acces la memorie data de la locația dorită se va încărca în registrul MDR (registrul de date al memoriei). Pe de altă parte, scrierea datelor în memorie se realizează în următoarele etape: se depune adresa în MAR, se depune data ce se vrea memorată în MDR şi *„se activează semnalul WRITE”* – la LC-3 WE devine 1. Scrierea registrului MAR poate fi făcută din trei surse (instrucțiuni *load*, *store* sau întrerupere software *TRAP*) prin intermediul diferitelor multiplexoare. Una din surse (ADDR1MUX din figura 6.17) este obținută din însumarea PC-ului cu IR[8:0] căruia i se extinde semnul pe 16 biți în cazul instrucțiunilor LD / ST şi LDI / STI. A doua sursă (MARMUX) este obținută din vectorul de întrerupere (câmpul IR[7:0] care se extinde cu 0-uri pe încă 8 poziții semnificative) pentru determinarea adresei primei instrucțiuni din rutina de tratare a întreruperii. A treia sursă (obținută tot prin intermediul multiplexorului ADDR1MUX din figura 6.17) este generată din suma dintre un registru general şi o valoare imediată pe 6 biți, extinsă cu 0-uri semnificative pe 16 biți, aferentă instrucțiunilor cu referire la memorie în mod de adresare indexat (LDR / STR).

Unitatea aritmetico-logică (ALU), componentă a *unității de procesare* şi evident a fluxului datelor la LC-3 ISA, acceptă ca şi intrări fie doi regiştrii sursă din **setul de regiştrii generali,** selectați cu câmpurile IR[8:6] şi respectiv IR[2:0] (SR1 şi SR2 în figura 6.17), fie un registru sursă şi valoarea imediată din câmpul registrului instrucțiunii căruia i se extinde semnul până la 16 biți. Multiplexorul SR2MUX în figura 6.17 va selecta între valoarea imediată şi registrul sursă cu ajutorul bitului 5 din registrul instrucțiunii. Rezultatul generat de către ALU va fi scris pe magistrala globală de unde va fi transferat în setul de regiştrii generali la locația specificată prin registrul destinație corespunzător instrucțiunii procesate. De asemenea, rezultatul va fi folosit de către structura „LOGIC” pentru a seta regiştrii booleeni de condiție (N, Z şi P). Pe lângă instrucțiunile aritmetico-logice (ADD, AND şi NOT) şi instrucțiunile de transfer date în regiştrii (LD, LDI, LDR şi LEA) setează codurile de condiție prin intermediul bus-ului global.

Setul de regiştrii generali, componentă esențială a fluxului datelor, este o structură *biport* pentru citire şi *uniport* pentru scriere. Scrierea în registrul destinație poate fi făcută de pe magistrala globală, informația ajungând aici fie de la ALU fie de la memorie prin registrul MDR.

Registrul **PC** furnizează prin intermediul magistralei globale registrului MAR adresa instrucțiunii de la care se va procesa (va începe următorul ciclu al instrucțiunii). PC-ul este încărcat prin intermediul unui multiplexor 4:1 (**PCMUX**) în funcție de instrucțiunea care se va executa

(salt condiționat, necondiționat, apel indirect, întrerupere software). Una din cele 4 intrări în PCMUX (vezi figura 6.17) este (PC+1), valoare stabilită în faza FETCH a instrucțiunii în curs de procesare. Dacă instrucțiunea este salt condiționat (salt care se face) atunci intrarea selectată în PCMUX va fi cea determinată din suma dintre PC şi câmpul IR[8:0] căruia i se extinde semnul de la 9 la16 biți. O altă intrare în PCMUX va fi obținută din setul de regiştrii generali (mai precis din R7 în cazul instrucțiunii RET – revenirea din subrutina utilizator). A patra intrare se obține prin intermediul magistralei globale datorită întreruperii software sau datorită instrucțiunii de salt / apel indirect (JSRR). Deşi în figura 6.17 se văd doar trei intrări, în realitate sunt cele patru anterior enunțate. Problema este că multiplexorul ADDR1MUX va selecta adevăratul responsabil pentru generarea noului PC – vechiul PC sau un registru general.

Unitatea de control îşi pune în aplicare numele, controlând în fiecare moment fluxul datelor în sistemul de calcul. După aducerea instrucțiunilor în IR, aceasta va fi decodificată (stabilită operația ce se va executa, operanzii). În fiecare ciclu maşină, prin intermediul automatului cu stări finite (FSM), comandată de biți ai registrului instrucțiunii, sunt stabilite semnalele de control pentru următoarea fază de procesare a instrucțiunii: cine devine conducătorul pe magistrala globală (*cine scrie informația*), care regiştrii vor fi scriși (WE va fi 1 pentru care structură logică), ce operație va efectua unitatea aritmetico-logică ?

6.3. EXERCIȚII ŞI PROBLEME

1. Se consideră cunoscute valorile inițiale pentru regiştrii şi locațiile de memorie următoare:

PC = 0x2081, R6 = 0x2035, LOC = 0x2044, Mem[0x2044] = 0x3456,
Mem[0x2041] = 0x12CF, Mem[0x3456] = 0x201F

Să se determine care este adresa efectivă a următoarelor instrucțiuni, precum şi codificarea fiecăreia dintre ele, specificând şi modurile de adresare:

a) LD R1, LOC
b) LDI R2, LOC
c) LDR R3, R6, #12

d) ADD R1, R3, R2
e) ADD R5, R1, #15

2. Să se identifice instrucțiunile a căror codificare este dată mai jos. Specificați conținutul fiecărui registru şi a locațiilor de memorie afectate după execuția instrucțiunilor. Ce se poate observa ?

PC	Codificare instrucțiuni	Semnificație
0x30F6	1110 001 000000100	
0x30F7	0001 010 001 1 01110	
0x30F8	0011 010 111111110	
0x30F9	0101 010 010 1 00000	
0x30FA	0001 010 010 1 00101	
0x30FB	0111 010 001 001110	
0x30FC	1010 011 011110100	

3. **a)** Secvența de cod de mai jos calculează expresia **(R1 × R1) + 6**, punând rezultatul în R3. Se cere pentru început să translatați programul din cod maşină (secvența binară a instrucțiunilor codificate) în limbaj de asamblare (sau eventual într-un limbaj codificat de tip transfer registru numit *RTL* – „*register transfer language*”) cunoscând instrucțiunile prezentate pe parcursul acestui capitol. De exemplu, instrucțiunea 1001 0000 0111 1111 este “R0 ← NOT R1” în RTL. După translatarea instrucțiunilor completați cu biții lipsă astfel încât programul să realizeze cerința impusă.

PC	Codificare instrucțiuni	Limbaj de asamblare / codificare de tip transfer registru
x3000	0101 0110 1110 0000	
x3001	_____ _____ _____ _____	
x3002	0001 0110 1100 0001	

x3003	0001 0100 1011 1111	
x3004	0000 101_ _____ _____	
x3005	0001 0110 1110 0110	

b) Dacă R1 şi R3 reţin numere întregi în complement faţă de 2 pe 16 biţi, pentru ce valori ale lui R1 rezultatul din R3 este corect. Justificaţi.

4. **a)** Instrucţiunile LC-3 următoare (A şi B) realizează în principiu acelaşi lucru. Explicaţi în cel mult 10 cuvinte ce fac ele.

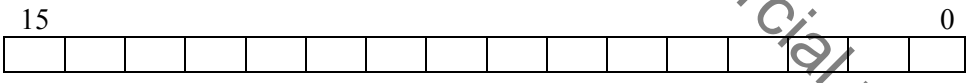
A 0000 0001 0101 0101

B 0001 0000 0010 0000

Totuşi, ele nu pot fi interschimbate deoarece nu fac chiar acelaşi lucru. În cel mult 10 cuvinte explicaţi care este diferenţa între cele două instrucţiuni.

b) În cadrul LC-3 ISA sunt adunate următoarele două numere întregi în complement faţă de 2 pe 16 biţi 0101011010101 şi 0011100111001111 obţinându-se rezultatul 1000111100100100. Rezultatul este corect ? Este vreo problemă ? Dacă *da* care este aceasta, dacă *nu* de ce nu este nici o problemă.

c) În cadrul LC-3 ISA se doreşte execuţia unei instrucţiuni care să scadă valoarea întreagă 20 din registrul R1 şi să depună rezultatul în R2. Poate fi realizat acest lucru ? Dacă *da* scrieţi mai jos codificarea instrucţiunii. Dacă nu explicaţi care este problema.



d) Ce specifică arhitectura setului de instrucţiuni aferentă unui procesor. Care sunt diferenţele dintre instrucţiunile LD, LDI şi LDR ? Dar diferenţa dintre acestea şi instrucţiunea LEA ?

e) Dacă dimensiunea totală a memoriei este de 1MB (2^{20} octeţi) şi adresabilitatea este de 32 de octeţi, câţi biţi de adresă sunt necesari pentru adresarea întregului spaţiu de memorie ?

5. Se consideră următorul program scris în limbaj maşină LC-3. Registrul R1 reprezintă variabila de intrare în această secvenţă de program iar R2

reprezintă variabila de ieşire la încheierea buclei. Dacă valoarea inițială a lui R1 este un număr întreg pozitiv *n*, exprimați ieşirea R2 în funcție de *n*. Se presupune că valorile prelucrate de acest program sunt suficient de mici pentru a nu apare depăşirea de reprezentare (*overflow*).

PC	Codificare instrucțiuni	Codificare de tip transfer registru (RTL)
x3001	0101 0100 1010 0000	R2 <- 0
x3002	0001 0100 1010 0001	R2 <- R2 + 1
x3003	0001 0100 1000 0010	R3 <- R2 + R2
x3004	0001 0100 1000 0011	R2 <- R2 + R3
x3005	0001 0010 0111 1111	R1 <- R1 - 1
x3006	0000 0011 1111 1100	BRp x3003

6. Rezolvați următoarele două cerințe:
a) Care este intervalul de numere întregi (exprimat în zecimal) care poate fi specificat în câmpul *Imm* (de valoare imediată) aferent unei instrucțiuni de adunare ADD ?
b) Scrieți codificarea a două instrucțiuni în LC-3 ISA care împreună decrementează registrul R3 cu 31 şi depune rezultatul în registrul R3. Completați următorul tabel.

PC	Codificare instrucțiuni	Limbaj de asamblare / codificare de tip transfer registru (RTL)
x3001		
x3002		

7. Se presupune că doriți să scrieți un program începând cu adresa 0x3001 şi care cuprinde instrucțiunile cu semantica din ultima coloană a tabelului următor:

PC	Codificare instrucțiuni	Codificare de tip transfer registru (RTL)
x3001		R2 <- M[R1+0]
x3002		R3 <- M[R1+1]
x3003		R4 <- NOT R3
x3004		R4 <- R4 + 1
x3005		R5 <- R2 + R4
x3006		BRzp x3009

x3007		M[R1+2] <- R3
x3008		BRnzp x3010
x3009		M[R1+2] <- R2

a) Completaţi tabelul cu codificarea fiecărei instrucţiuni.

b) Parcurgeţi programul pas cu pas începând cu adresa 0x3001 şi completaţi conţinutul următoarelor resurse pe măsură ce se execută fiecare instrucţiune. Prima coloană reprezintă adresa instrucţiunilor ce vor fi procesate, a doua coloană ilustrează operaţiile propriu-zis de executat. Următoarele colane redau starea iniţială a regiştrilor generali şi a celor de condiţie din LC-3 ISA. Se presupune de asemenea că la adresa de memorie 0x3100 se găseşte valoarea 14 iar la adresa 0x3101 se găseşte valoarea 27.

PC	Operaţia	R0	R1	R2	R3	R4	R5	R6	R7	CCs
Starea iniţială a regiştrilor ➔		0	x3100	0	3	4	5	6	7	
x3001	R2 ← M[R1+0]			14						P
x3002										
x3003										
x3004										
x3005										
x3006										
x3007										
x3008										
x3009										

c) Într-o singură propoziţie ce calculează programul anterior?

8. Tabelul următor conţine un program LC-3 codificat binar şi stocat în memorie începând cu adresa 0x3001.

PC	**Codificare instrucţiuni** 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	**Codificare de tip transfer registru (RTL)**
x3001	0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0	R2 ← M[R1+0]
x3002	0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 1	
x3003	1 0 0 1 0 1 1 0 1 1 1 1 1 1 1 1	
x3004	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	

x3005	0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 1	
x3006	0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0	
x3007	1 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	
x3008	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	
x3009	0 1 1 1 0 1 0 0 0 1 0 0 0 0 1 0	

a) Pentru început determinați ce face fiecare instrucțiune. Descrieți printr-o codificare de tip transfer registru (RTL).

b) Parcurgeți programul pas cu pas începând cu adresa 0x3001 şi completați conținutul următoarelor resurse pe măsură ce se execută fiecare instrucțiune. Prima coloană reprezintă adresa instrucțiunilor ce vor fi procesate, a doua coloană ilustrează operațiile propriu-zis de executat (determinate la punctul anterior). Următoarele colane redau starea inițială a regiştrilor generali şi a celor de condiție din LC-3 ISA. Se presupune de asemenea că la adresa de memorie 0x3100 se găseşte valoarea -34 iar la adresa 0x3101 se găseşte valoarea -20.

PC	Operația	R0	R1	R2	R3	R4	R5	R6	R7	CCs
Starea inițială a regiştrilor ➔			x3100	2	3	4	5	6	7	
x3001	R2 ← M[R1+0]			-34						N
x3002										
x3003										
x3004										
x3005										
x3006										
x3007										
x3008										
x3009										

c) Într-o singură propoziție ce calculează programul anterior ?

9. LC-3 ISA nu furnizează un opcode pentru funcția logică SAU (OR). Cu toate acestea, se poate scrie o secvență scurtă de instrucțiuni pentru implementarea operației de SAU logic. Cele patru instrucțiuni de mai jos efectuează operația de SAU logic având ca operanzi sursă regiştrii R1 şi R2 iar rezultatul este depus în registrul R3. Completați cele două instrucțiuni care lipsesc pentru obținerea rezultatului dorit.

(1): 1001 100 001 111111

(2): _____

(3): 0101 110 100 000 101

(4): _____

10. a) Dispunând doar de instrucţiunile ADD, AND şi NOT cum poate fi implementată o instrucţiune de scădere (descrieţi succesiunea de instrucţiuni care concură la execuţia unei operaţii de scădere):

SUB Reg[Dst], Reg[Src1], Reg[Src2].

b) Aceeaşi problemă dar pentru instrucţiunea

XOR Reg[Dst], Reg[Src1], Reg[Src2].

c) Cum copiem un registru sursă într-unul destinaţie:

MOVE Reg[Dst], Reg[Src1].

d) Cum iniţializăm un registru cu 0 ?

11. Pe timpul cărei faze de procesare din ciclul instrucţiunii se modifică registrul PC ? Dar IR, MAR şi MDR ? Exemplificaţi pe următoarele cazuri:

a) Pentru instrucţiunea de adunare al cărei opcode este 0001 (ADD):

	Fetch Instrucţiune	Decodificare	Evaluare Adresă	Fetch Operand	Execuţie	Scrie Rezultat
PC						
IR						
MAR						
MDR						

b) Pentru instrucţiunea de citire din memorie în mod de adresare direct al cărei opcode este 0010 (LD):

	Fetch *Instrucțiune*	Decodificare	Evaluare Adresă	Fetch Operand	Execuție	Scrie Rezultat
PC						
IR						
MAR						
MDR						

c) Pentru instrucțiunea de scriere în memorie în mod de adresare direct al cărei opcode este 0011 (ST):

	Fetch *Instrucțiune*	Decodificare	Evaluare Adresă	Fetch Operand	Execuție	Scrie Rezultat
PC						
IR						
MAR						
MDR						

d) Pentru instrucțiunea de salt condiționat al cărei opcode este 0000 (BR):

	Fetch *Instrucțiune*	Decodificare	Evaluare Adresă	Fetch Operand	Execuție	Scrie Rezultat
PC						
IR						
MAR						
MDR						

12. Pentru următoarele instrucțiuni descrieți ce operații au loc pe perioada fiecărei faze de procesare din ciclul instrucțiunii.
a) ADD R1, R2, R3
b) LD R1, LABEL
c) NOT R1, R1

13. Ce se întâmplă dacă în codificarea instrucțiunii de salt BR x, *Label*, câmpul IR[11:9] – este „000” ? Dar dacă este „111” ?

7. LIMBAJUL DE ASAMBLARE AFERENT ARHITECTURII LC-3. ASAMBLORUL. ETAPELE GENERĂRII CODULUI MAŞINĂ. TABELA DE SIMBOLURI

7.1. MOTIVE PENTRU A PROGRAMA ÎN LIMBAJ DE ASAMBLARE

Limbajul de asamblare este un limbaj de programare, principala sa deosebire față de limbajele de nivel înalt, cum sunt BASIC, PASCAL şi C/C++, fiind aceea că el oferă doar câteva tipuri simple de comenzi şi date. Limbajele de asamblare nu specifică tipul valorilor păstrate în variabile, lăsând programatorul să aplice asupra lor operațiile potrivite. Scopul limbajului de asamblare este de a face mai prietenos procesul de programare decât programarea directă în limbaj maşină.

În ciuda unor dezavantaje clare, specifice programării în limbaj de asamblare: ***lizibilitate greoaie a codului*** (greu de scris, citit şi înțeles), ***depanare şi întreținere dificilă, lipsa portabilității codului***, există motive pentru care încă, nu s-a renunțat la limbajul de asamblare. Printre acestea se numără viteza ridicată de execuție a programelor, spațiul relativ redus consumat de acestea dar şi permiterea accesului la anumite resurse hardware, acces care nu este disponibil în limbajele de nivel înalt [Seb].

Un motiv ***pentru*** **programarea în limbaj de asamblare** îl reprezintă prezența rutinelor aferente sistemului de operare. Întrucât acestea sunt apelate foarte des pe parcursul funcționării sistemului de calcul (atât în programe utilizator cât şi în funcții ale sistemului de operare), este necesar un **timp redus de execuție** din partea lor şi să ocupe un **cât mai puțin spațiu de memorie**. Programarea în limbaj de asamblare este în primul rând dependentă de maşina (microarhitectura hardware) pe care se procesează, obligând programatorul să cunoască cât mai detaliat arhitectura setului de instrucțiuni – ISA.

Un alt motiv ***în favoarea*** programării în limbaj de asamblare îl reprezintă amploarea şi dinamismul existent în dezvoltarea de sisteme

dedicate. Performanța se focalizează pe îndeplinirea cerințelor de timp real ale aplicației (aplicații cu microcontrolere, procesoare de semnal, aparate foto şi camere video, comanda aparatelor electrocasnice, telefoane mobile, imprimante, comenzi auto, jocuri electronice, switch-uri pentru rețele, etc). Sunt caracterizate în principal de **consumuri reduse de putere** (deseori sunt alimentate prin baterii şi acumulatori) şi **memorii de capacități relativ reduse**.

Un alt aspect care ***recomandă studiul limbajelor de asamblare*** (deşi este dependent de un anumit ISA principiile sunt aceleaşi) îl reprezintă **caracterul său formativ**: o cunoaştere şi o mai bună înțelegere a modului de lucru al procesorului, a modurilor de adresare, a organizării memoriei, a lucrului cu stiva poate ajuta la înțelegerea mecanismelor de generare a stivei de date aferentă funcțiilor, a recursivității şi a transferului de parametrii la nivelul funcțiilor, conducând în final la scrierea de programe eficiente în limbajele de nivel înalt (independente de maşină) [Seb, Lun05, Mus97].

De asemenea, ***medii integrate de programare-dezvoltare*** şi compilatoare de limbaje de nivel înalt (C, Pascal, etc.) ***prezintă facilități de inserare în codul sursă de nivel înalt a liniilor scrise direct în limbaj de asamblare*** sau link-editarea într-un singur modul a mai multor module obiect provenite de la compilarea unor coduri sursă, scrise în limbaje de programare diferite (C, asamblare). În cazul aplicațiilor ample, modulele care necesită structuri de date complexe şi tehnici de programare / algoritmi complicați de rezolvare, sunt scrise module în limbaje de nivel înalt, iar cele care sunt critice din punct de vedere al timpului de execuție şi al resurselor utilizate sunt implementate în limbaj de asamblare specific procesorului pe care se execută aplicația.

La toate acestea se mai pot adăuga cauze externe, cum ar fi integrarea programatorului într-un colectiv / proiect software care lucrează la rutine ale sistemului de operare sau care trebuie să modifice anumite programe existente, scrise în asamblare, sau anumite zone de cod rulează prea încet sau consumă prea multă memorie.

Pe lângă prezentarea instrucțiunilor din limbajul de asamblare LC-3 acest capitol urmăreşte descrierea etapelor necesare generării imaginii executabile a unui program pornind de la sursa sa scrisă în limbaj de asamblare. Instrumentul software care realizează acest lucru se numeşte **asamblor**. Acesta **transformă fiecare instrucțiune scrisă în limbaj de asamblare** (într-o formă „cât de cât" prietenoasă programatorului – de exemplu ADD R6,R2,R1) **într-o singură instrucțiune scrisă în limbaj maşină** (succesiune de 0 şi 1 – extrem de prietenoasă procesorului şi întregului sistem de calcul 0001110010000001). Procesul de asamblare

realizează o corespondență între fiecare simbol (element de limbaj asamblare) şi setul de instrucțiuni specificat de ISA-ul maşinii hardware pe care se execută programul. Mnemonicile (ADD, LDR, BRnz, etc.) sunt înlocuite cu biții câmpului de opcode, etichetele – nume simbolice (LOOP, YES, NO, AGAIN, etc.) – sunt înlocuite cu adresele reale ale locațiilor de memorie. Suplimentar, datorită directivelor de asamblare se fac operații de alocare de memorie şi inițializare date. Trebuie adăugat că unei instrucțiuni dintr-un limbaj de nivel înalt (ex: C, Pascal, etc.) îi corespunde o secvență de instrucțiuni maşină (unul sau mai multe coduri).

7.2. SINTAXA ASAMBLOR LC-3 [Patt03]

Fiecare linie într-un program asamblare reprezintă una din următoarele:

- instrucțiune
- directivă de asamblare
- comentariu

Spațiile albe dintre simboluri sunt ignorate. Limbajul de asamblare **LC-3 nu este case senzitiv**. Comentariile în fişiere de asamblare LC-3 încep cu simbolul '**;**'(ca şi la INTEL). Orice urmează acestui caracter până la sfârşitul liniei este ignorat. Identificatorii sunt o secvență de caractere alfanumerice, linie de subliniere, şi trebuie să nu înceapă cu un număr. Opcode-ul instrucțiunilor sunt cuvinte rezervate care nu pot fi folosite ca identificatori. Etichetele sunt declarate prin aşezarea lor la începutul unei linii. Numerele sunt implicit în baza 10 şi sunt precedate de simbolul #. Dacă sunt precedate de caracterul 0x, ele sunt interpretate în sistemul de numerație hexazecimal. Deci, #256 şi 0x100 semnifică aceeaşi valoare.

Şirurile sunt încadrate de ghilimele '**"**'. Caracterele speciale din şiruri urmează convenția limbajului C. Astfel:

- linie nouă \n
- tab \t
- ghilimele \"

Directivele de asamblare reprezintă false operații întrucât ele nu referă operații executate de programul utilizator, fapt pentru care nu sunt translatate în instrucțiuni ale limbajului maşină (LM). Deşi se aseamănă cu instrucțiunile (caracterizate de un cod de operație) ele sunt precedate întotdeauna de „**.**" (simbolul punct). Sunt utilizate de asamblor pentru a

aloca, iniţializa zone de memorie de date necesare în program dar şi pentru a marca începutul şi sfârşitului zonei de cod alocate programului sursă.

În tabelul 7.1. sunt prezentate, câteva din **directivele de asamblare ale LC-3**.

Nume directivă	Operand	Semantică
.ORIG	**address**	Adresa de start a programului în memorie. Această adresă se va încărca în PC la lansarea în execuţie a codului maşină.
.END		Marchează sfârşitul programului sursă. Atenţionează asamblorul să nu mai incrementeze contorul de linii în etapa de generare a tabelei de simboluri.
.BLKW	**n**	Alocă *n* cuvinte de spaţiu pentru eventuale salvări.
.EXTERNAL		Eticheta declarată EXTERNAL indică asamblorului faptul că este definită în alt modul (tabela de simboluri aferentă respectivului modul îi asignează o adresă).
.FILL	**val**	Alocă un cuvânt de memorie şi îl iniţializează cu valoarea *val*.
.STRINGZ	**n - *character string***	Alocă n+1 locaţii de memorie (cuvinte) şi salvează în această zonă şirul de caractere respectiv (câte un cuvânt per locaţie). În locaţia n+1 este salvat caracterul x0000 – terminatorul NULL – care marchează sfârşitul şirului.

Tabelul 7.1. Directivele de asamblare ale LC-3

Formatul instrucţiunilor aferente LC-3 ISA este de lungime fixă (caracteristică a procesoarelor RISC) pe 16 biţi, şi după cum poate fi observat şi în figura 7.1 este alcătuit din 4 câmpuri: **OPCODE**, **OPERANZI, ETICHETE** şi **COMENTARII** (dintre care primele două sunt obligatorii iar ultimele două sunt opţionale)**.**

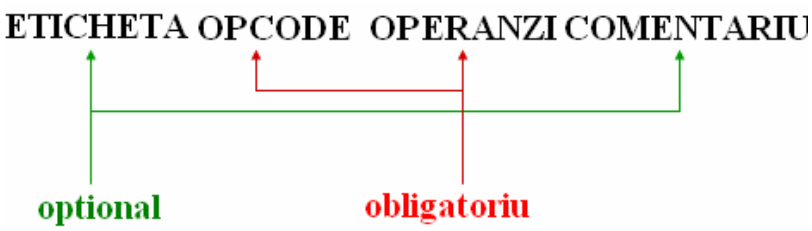


Figura 7.1. Formatul instrucţiunilor în limbajul de asamblare LC-3

Opcode – ul reprezintă cuvinte rezervate care nu trebuie să fie folosite pe post de identificatori, etichete şi care corespund instrucţiunilor din setul LC-3 ISA (în limbaj maşină opcode-ul este reprezentat pe 4 biţi). Toate instrucţiunile şi implicit opcode-urile sunt ilustrate în subcapitolul 7.2.1 (ex: ADD, AND, LD, LDR, …).

Operanzii pot exista sub următoarea formă:

- **regiştrii** – specificaţi prin **Rn**, unde ***n*** reprezintă numărul registrului ($n \in [0 \div 7]$ la LC-3 ISA)
- **valori numerice** – reprezentate în sistemul de numeraţie zecimal folosind # sau hexazecimal (folosind *0x*)
- **etichete** – nume simbolice a unor locaţii de memorie (vezi LD, ST, BR etc.)

Operanzii sunt separaţi prin virgulă. Numărul, ordinea şi tipul corespund formatului instrucţiunii (vezi subcapitolul 7.2.1).

```
Ex:     ADD     R1, R1, R3      R1←R1+R3
        ADD     R1, R1, #3      R1←R1+3
        LD      R6, NUMBER      R6←MEM[NUMBER]
        BRz     LOOP            Execută salt la LOOP pe condiţie
                                egală cu zero.
```

Etichetele sunt plasate la începutul fiecărei linii şi asignează un nume simbolic corespunzător unei linii de adresă (folosită apoi de instrucţiunile de salt sau de instrucţiunile cu referire la memorie).

```
Ex1     LOOP    ADD     R1,R1,#-1
                BRp     LOOP

Ex2:            .
                .
                LD      R2, SIX
                .
                .
        SIX
                .FILL   6
```

Comentariu este orice secvenţă de simboluri care urmează caracterului „**;**”. Este ignorat de către asamblor, rolul său fiind de a ajuta dezvoltatorii şi utilizatorii de programe pentru o mai bună documentare şi înţelegere a secvenţelor de cod. Există însă şi câteva artificii (reguli de respectat) în cazul folosirii comentariilor:

- Trebuie evitată comentarea unor lucruri evidente: de ex. "se decrementează R1".
- Furnizaţi indicii suplimentare de genul "în R6 se calculează produsul prin acumulare – însumări repetate".
- Folosiți comentariile pentru a separa bucățile de program cu caracter distinct.

În continuare sunt prezentate câteva sfaturi legate de stilul de programare pentru îmbunătățirea lizibilității şi înțelegerii codului scris în limbaj de asamblare, dar nu numai.

1. Realizați un antet cu numele autorului, eventuale date de contact şi scopul programului
2. Etichetele, opcode-ul, operanzii şi comentariile să înceapă în aceeaşi coloană, exceptând cazurile în care întreg rândul este un comentariu.
3. Folosiți comentarii pentru a explica ce reține fiecare registru şi respectiv ce face fiecare instrucțiune.
4. Folosiți nume simbolice edificatoare, dacă este cazul mixați literele mici cu cele mari (de ex. ASCIItoBinary, InputRoutine, SaveR1).
5. Folosiți comentariile pentru a separa secțiunile de program cu caracter distinct.
6. Evitați trunchierile de instrucțiuni sau desfăşurarea acestora pe două pagini de text. Enunțurile lungi separați-le într-o manieră estetică.

Primul program scris în limbaj de asamblare LC-3 propus drept exemplu realizează înmulțirea unui număr întreg cu o constantă (6) prin adunări repetate.

```
        .ORIG   x3050
        LD      R1, SIX
        LD      R2, NUMBER
        AND     R3, R3, #0      ; Se inițializează R3 cu 0. În R3 se va
                                  calcula produsul.

; Bucla de calcul a sumei repetate (produsul)

AGAIN
        ADD     R3, R3, R2      ; R3←R3+R2
        ADD     R1, R1, #-1     ; R1 păstrează numărul de iterații (de
                                  însumări care trebuie făcute) R1←R1 - 1
```

```
        BRp     AGAIN    ; dacă este setat bitul de condiţie P (în
                           urma scăderii rezultatul
                         ; este strict pozitiv) se execută salt la
                         eticheta AGAIN şi reia o
                         ; nouă iteraţie
        HALT
NUMBER
        .BLKW   1
; pentru ca programul să execute înmulţirea dintre numerele dorite trebuie ca
; la locaţia NUMBER să se intervină la nivelul simulatorului şi să se înscrie
; o valoare numerică sau directiva de asamblare .BLKW 1 să fie înlocuită cu
; una de genul .FILL număr_dorit

SIX
        .FILL   x0006
        .END
```


7.2.1. CORESPONDENŢA LIMBAJ DE ASAMBLARE LC-3 – LIMBAJ MAŞINĂ SPECIFIC LC-3 ISA

M	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	**0**	**0**	**0**	**1**	**DR**			**SR1**			**0**	**0**	**0**	**SR2**		
ADD*	**0**	**0**	**0**	**1**	**DR**			**SR1**			**1**	**Imm5**				
AND	**0**	**1**	**0**	**1**	**DR**			**SR1**			**0**	**0**	**0**	**SR2**		
AND*	**0**	**1**	**0**	**1**	**DR**			**SR1**			**1**	**Imm5**				
BR	**0**	**0**	**0**	**0**	**n**	**z**	**p**	**Pageoffset9**								
JSR	**0**	**1**	**0**	**0**	**L**	**0**	**0**	**Pageoffset9**								
JSRR	**1**	**1**	**0**	**0**	**L**	**0**	**0**	**BaseR**			**Index6**					
LD	**0**	**0**	**1**	**0**	**DR**			**Pageoffset9**								
LDI	**1**	**0**	**1**	**0**	**DR**			**Pageoffset9**								
LDR	**0**	**1**	**1**	**0**	**DR**			**BaseR**			**Index6**					
LEA	**1**	**1**	**1**	**0**	**DR**			**Pageoffset9**								

NOT	**1**	**0**	**0**	**1**	**DR**			**SR1**			**1**	**1**	**1**	**1**	**1**	**1**
RET	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
RTI	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ST	**0**	**0**	**1**	**1**	**SR**			**Pageoffset9**								
STI	**1**	**0**	**1**	**1**	**SR**			**Pageoffset9**								
STR	**0**	**1**	**1**	**1**	**SR**			**BaseR**			**Index6**					
TRAP	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**Trapvect8**							

Legendă:

Coloana **M** reprezintă mnemonicele instrucțiunilor limbajului de asamblare LC-3.

Corespunzător fiecărei linii (instrucțiuni) coloanele **15÷0** reprezintă instrucțiunea în limbaj maşină, biții 15÷12 reprezentând **opcode-ul** (se reaminteşte faptul că LC-3 ISA conține 16 instrucțiuni distincte).

DR – registru destinație

SR – registru sursă

Imm5 – valoare imediată pe 5 biți. Instrucțiunile de adunare şi „ŞI Logic" pot opera atât cu doi operanzi aflați în regiştrii cât şi cu unul situat în registru şi al doilea fiind o valoare imediată căreia i se va extinde semnul pe 16 biți.

BaseR – registru de bază folosit în instrucțiunile cu referire la memorie şi mod de adresare indexat iar **Index6** reprezintă deplasamentul pe 6 biți.

Pageofset9 – deplasament pe 9 biți relativ la o adresă de bază (aceasta poate fi PC-ul instrucțiunii următoare în cazul instrucțiunilor LD, ST, BR sau JSR).

Trapvect8 – codificare vectorului de întrerupere pe 8 biți fără semn, existând 256 de întreruperi la arhitectura LC-3 ISA

7.2.2. APELURI SISTEM

Întreruperile software la nivel *low* numite **apeluri sistem** reprezintă un mic set de servicii ale sistemului de operare prin instrucțiuni (*syscall* – la procesorul MIPS, întreruperi software *int 21h* la Intel, *TRAP* la arhitectura

LC-3). Pentru a apela un serviciu, trebuie încărcat codul apelului sistem într-unul din regiştrii arhitecturali (la MIPS în $2 ($v0), la LC-3 în R0, la Intel în AH). În tabelul 7.2 sunt prezentate cele mai uzuale apeluri sistem implementate în cadrul arhitecturii LC-3 - ISA.

Serviciul	Cod Apel Sistem	Descriere
HALT	**TRAP x25**	Se opreşte execuţia şi se afişează un mesaj corespunzător pe consolă.
IN	**TRAP x23**	Se afişează un mesaj pe consolă – „gen prompter", citeşte (cu ecou) un caracter de la tastatură.. Codul ASCII al caracterului respectiv este memorat în registrul R0[7:0].
OUT	**TRAP x21**	Afişează pe consolă caracterul având codul ASCII in R0[7:0].
GETC	**TRAP x20**	Citeşte (fără ecou) un caracter de la tastatură.. Codul ASCII al caracterului respectiv este memorat în registrul R0[7:0].
PUTS	**TRAP x22**	Afişează pe consolă un şir de caractere încheiat cu terminatorul NULL. Adresa şirului se găseşte în R0.

Tabelul 7.2. Servicii Sistem

7.3. PROCESUL DE ASAMBLARE – ETAPELE GENERĂRII CODULUI MAŞINĂ

Procesul de asamblare presupune convertirea unui fişier scris în mnemonică de asamblare (.asm – la LC-3 şi Intel, .s la MIPS) într-un fişier executabil pe platforma destinaţie (în cazul de faţă simulatorul LC-3). Codul generat poate fi direct executabil (gata de rulare) sau în format cod obiect (pseudo-executabil .obj) care trebuie legat (link-editat) cu codul obiect al funcţiilor de bibliotecă apelate în modulul respectiv sau cu alte fişiere obiect. Se disting două etape:

- Scanarea codului sursă asamblare şi **generarea tabelei de simboluri**

- Reanalizarea programului sursă şi folosind informaţiile din tabela de simboluri se **generează codul în limbaj maşină specific maşinii destinaţie**

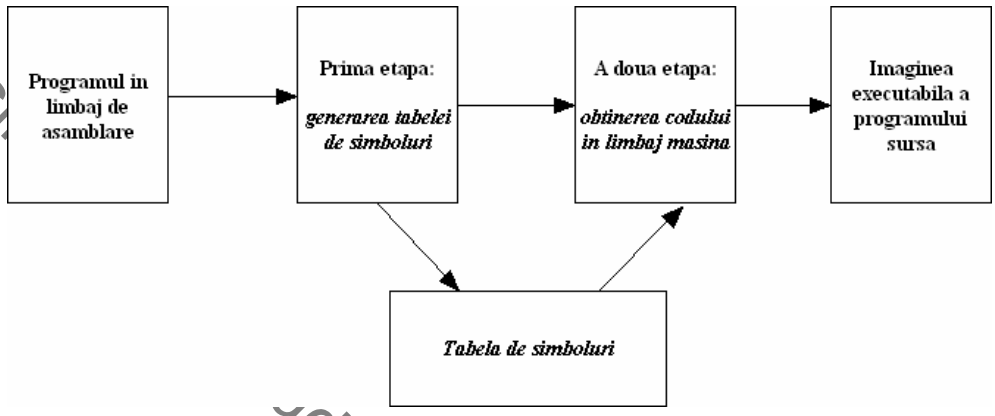


Figura 7.2. Etapele generării codului maşină pornind de la limbajul de asamblare LC-3

7.3.1. GENERAREA TABELEI DE SIMBOLURI (TS)

Reprezintă prima etapă a procesului de asamblare şi presupune următoarea secvenţă de paşi:

- Se parcurge fiecare linie din fişierul sursă asamblare.
- Pentru fiecare linie nevidă din program, se determină dacă aceasta conţine o etichetă (pe prima coloană) şi se calculează adresa corespunzătoare (vezi mai jos) adăugându-se la *tabela de simboluri*. O linie care conţine doar comentariu se consideră vidă din punctul de vedere al asamblorului.
- Se identifică directiva **.ORIG**, care stabileşte adresa de start a primei instrucţiuni.
- Se păstrează un contor local (LC) al adreselor (incrementat cu 1 la fiecare instrucţiune). Contorul local se iniţializează cu valoarea adresei specificată prin directiva .ORIG.
- Procesul se încheie la întâlnirea directivei **.END**.
- Trebuie specificat că la întâlnirea directivelor de asamblare **.BLKW** sau **.STRINGZ** contorul local se incrementează cu numărul de cuvinte (words) alocate.

Revenind la exemplul din subcapitolul 7.2 tabela de simboluri arată în felul următor:

Simbol	Adresă
AGAIN	0x3053
NUMBER	0x3057
SIX	0x3058

7.3.2. OBŢINEREA CODULUI ÎN LIMBAJUL MAŞINĂ

Reprezintă cea de-a doua etapă a procesului de asamblare şi presupune parcurgerea de la început a programului asamblare, linie cu linie, şi folosind(dacă este cazul) informaţiile din tabela de simboluri tocmai generată se translatează fiecare instrucţiune în cod binar (limbaj maşină). Practic fiecare opcode este înlocuit cu secvenţa de 4 biţi corespunzătoare (vezi secţiunea 7.2.1) iar cei 8 regiştri sunt înlocuiţi cu codificarea binară pe 3 biţi a indexului lor. La întâlnirea unei etichete care face parte din corpul unei instrucţiuni (nu este pe prima coloană) se caută aceasta în tabela de simboluri şi se ia adresa corespunzătoare. Se verifică să aparţină această adresă în intervalul +256 / -255 linii faţă de instrucţiunea în cauză (într-o pagină de 9 biţi cu semn relativă la PC-ul curent). Se înlocuieşte apoi în câmpul corespunzător instrucţiunii curente cei 9 biţi mai puţini semnificativi ai adresei obţinute făcând diferenţa dintre **adresa din TS şi PC-ul instrucţiunii următoare** (celei cu eticheta).

Pot apare următoarele probleme:

- Folosirea necorespunzătoare a numărului sau tipului de argumente. De exemplu:

```
NOT     R1,#7
ADD     R1,R2
ADD     R3,R3,NUMBER
```

- Folosirea argumentelor de tip *imediat* care depăşesc domeniul de reprezentare. De exemplu:

```
ADD     R1, R2, #1023
```

- Utilizarea adreselor (asociate etichetelor) care nu se regăsesc în aceeaşi pagină de memorie cu instrucţiunea curentă. În acest caz nu se poate aplica modul direct de adresare (trebuie utilizate instrucţiuni cu mod de adresare indexat).

- De asemenea pot apărea erori dacă se folosesc adrese nedeclarate extern şi care nu se găsesc în tabela de simboluri.

Revenind la exemplul din subcapitolul 7.2, ţinând cont şi de tabela de simboluri generată în subcapitolul 7.3.1, codul în limbaj maşină al programului va arăta astfel:

```
; Program de înmulțire a unui număr cu 6
                .ORIG    x3050
x3050           LD R1, SIX                  x3050   0010001000000111
x3051           LD R2, NUMBER               x3051   0010010000000101
x3052           AND  R3, R3, #0             x3052   0101011011100000
                ; Bucla de calcul
x3053   AGAIN   ADD      R3, R3, R2         x3053   0001011011000010
x3054           ADD      R1, R1, #-1        x3054   0001001001111111
x3055           BRp      AGAIN              x3055   0000001111111101
x3056           HALT                        x3056   1111 0000 0010 0101
x3057   NUMBER          BLKW    1           x3057   0000000000000000
x3058   SIX     .FILL    x0006              x3058   0000 0000 0000 0110
                .END
```

După cum se poate observa în figura 7.3, în urma aplicării procesului de asamblare unui fişier sursă asamblare (**.asm**), folosind opţiunea de **Assemble** din meniul **Translate** al utilitarului **LC-3Edit**, pe lângă fişierul obiect (.obj) sunt generate următoarele fişiere auxiliare:

- Fişierul binar (**.bin**) conţine translatarea fiecărei instrucţiuni în binar (limbaj maşină) inclusiv linia corespunzătoare directivei .ORIG.
- Fişierul binar (**.hex**) este copia semantică a fişierului binar cu deosebirea că liniile sunt de fapt nu valori în binar ci în hexazecimal.
- Fişierul (**.sym**) reprezintă tabela de simboluri (ilustrează maparea **nume_eticheta → Adresă**).
- Fişierul (**.lst**) conţine pe prima coloană adresa instrucţiunii care se va procesa, pe următoarea codificarea hexazecimală a instrucţiunii, urmată de codificarea binară a instrucţiunii, numărul curent al liniei (în zecimal) şi pe ultima coloană instrucţiunea în mnemonică asamblare.
- Fişierul obiect (**.obj**) conţine aceleaşi informaţii ca şi fişierul (.**bin**) cu deosebirea că informaţia (programul în limbaj maşină) în fişierul obiect este scrisă în **format binar** (date în format neprintabil) în timp ce în fişierul .bin informaţia este stocată în format ASCII (poate fi vizualizată folosind un editor de text gen *notepad.exe*). Fişierul .obj urmează a fi încărcat în simulatorul LC-3 şi executat. Considerând un nivel suficient

de redus de abstractizare, formatul binar de fişier este caracterizat de o reprezentare compactă a informaţiei în vederea unei analize sintactice şi semantice eficiente („*parsing*"). Formatul **ELF („*Executable and Linking Format*")** reprezintă un format binar implicit de reprezentare a fişierelor în sistemul de operare Linux. Fişierul ELF este compus dintr-un antet („ELF header") care conţine detalii despre tipul fişierului obiect – direct executabil, partajabil, relocabil, despre arhitectura ţintă pe care se va procesa, etc.; antetul este urmat de 0 sau mai multe segmente şi 0 sau mai multe secţiuni. Segmentele conţin informaţii necesare pentru execuţia fişierului de aplicaţie, în timp ce secţiunile conţin date importante necesare procesului de creare a legăturilor între mai multe fişiere obiect (link-editare) şi eventual pentru relocare (vezi în continuare).

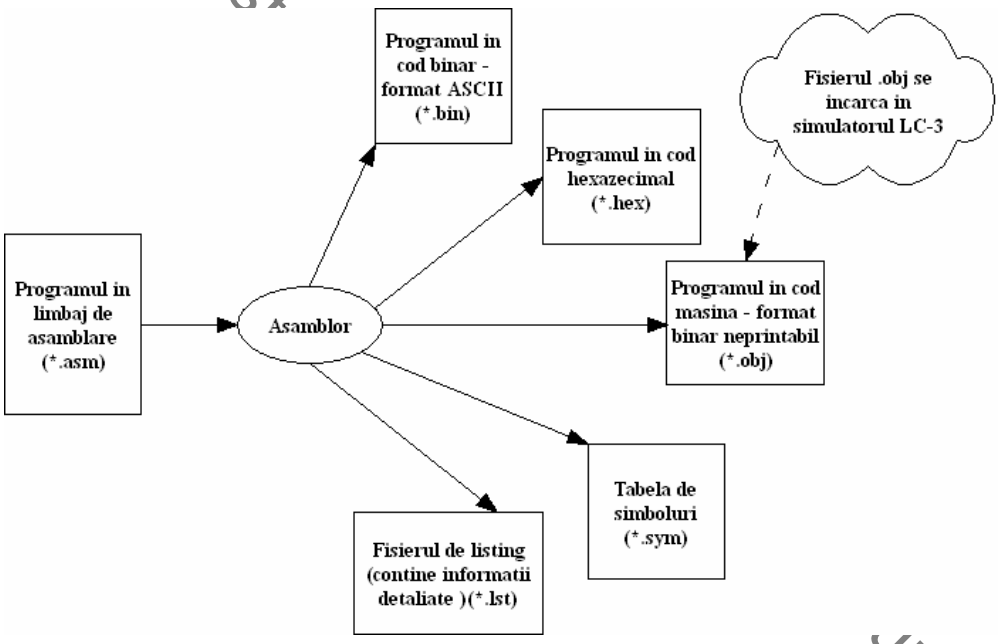


Figura 7.3. Fişierele generate de asamblorul inclus în LC-3Edit în urma asamblării unui cod sursă

7.3.3. CREAREA LEGĂTURILOR ÎNTRE MODULE, ÎNCĂRCAREA ŞI LANSAREA ÎN EXECUȚIE

Un fişier obiect nu este neapărat un program complet, gata de rulare. **Imaginea executabilă poate fi generată din mai multe module obiect**: programe de aplicație asamblate, scrise de unul sau mai mulți utilizatori şi respectiv funcții de bibliotecă furnizate de către sistemul de operare. Simulatorul LC-3 permite încărcarea mai multor fişiere obiect în memorie şi începerea execuției de la o anumită adresă. Un singur fişier obiect trebuie să fie modulul principal (identic cu cazul proiectelor în mediile de programare bazate pe limbajul C când există un singur modul ce conține funcția *main*). Rutinele sistem, destinate interfeței cu tastatura sau consola, sunt încărcate automat în *memoria sistem*, la adrese inferioare valorii 0x1000 – vezi harta de memorie la arhitectura LC-3 (capitolul 10 - figura 10.4). Prin convenție, codul utilizator (instrucțiunile) poate fi încărcat în intervalul de adrese 0x3000 şi 0xCFFF. Fiecare fişier obiect trebuie să includă o adresă de start. Programatorii trebuie să aibă grijă să nu suprapună mai multe programe obiect în memorie iar referințele declarate prin directiva .EXTERNAL (încrucişate între mai multe module) trebuie rezolvate atent la generarea codului obiect.

Crearea legăturilor – „*Linking*" – reprezintă procesul de rezolvare a simbolurilor partajate între module independente. Link-editorul reprezintă instrumentul software (de regulă component al compilatorului) care va căuta în tabela de simboluri a tuturor modulelor obiect generate pentru a afla adresa reală a etichetelor declarate .EXTERNAL în anumite module şi de încheia faza de generare de cod, pregătind încărcarea şi apoi execuția.

Încărcarea şi începerea execuției – „*Loading"* – reprezintă procesul de copiere a imaginii executabile a unui program în memorie (se copiază lista de instrucțiuni maşină cu toate referințele simbolice rezolvate în memorie, începând cu adresa specificată prin directiva .ORIG). Lansarea în execuție se face prin copierea adresei de start în registrul PC (*program counter*). „*Încărcătoare software*" mai complexe sunt capabile să **reloce** (gestioneze eficient, modificând chiar) spațiul aferent imaginii executabile astfel încât codul maşină să poată fi încadrat în memoria disponibilă. Problema relocării apare mai ales în situația utilizării mai multor fişiere obiect scrise de programatori independenți şi care folosesc adrese de start identice sau situate într-o vecinătate a spațiului de memorie. În aceste cazuri trebuie reajustate target-urile instrucțiunilor de ramificație şi adresele datelor solicitate de către instrucțiunile cu referire la memorie (load / store).

7.4. EXERCIŢII ŞI PROBLEME

1. Secvenţa de program de mai jos urmăreşte să deplaseze registrul R3 la stânga cu 4 biţi, dar conţine o eroare. Identificaţi eroarea şi explicaţi cum poate fi înlăturată astfel încât programul să funcţioneze corect.

```
        .ORIG   x3000
        AND     R2, R2, #0
        ADD     R2, R2, #4
LOOP    BRz     DONE
        ADD     R2, R2, # -1
        ADD     R3, R3, R3
        BRnzp   LOOP
DONE    HALT
        .END
```

2. Următorul program intenţionează să citească două numere (caractere) de la tastatură, să le adune şi apoi să afişeze suma.

```
ADDTHEM     TRAP   x23        ; IN – se citeşte primul număr
            ADD    R4,R0,#0
            TRAP   x23        ; IN –  se citeşte al doilea număr
            ADD    R0,R4,R0
            TRAP   x21        ; OUT – afişare rezultat
            TRAP   x25    ; HALT – încheiere program utilizator
```

a) Ce nu este în regulă cu acest program ?
b) Dacă se introduce de la tastatură 5, şi apoi 3, ce rezultat se va afişa pe consolă (ecran) ?

3. Câte locaţii de memorie va aloca asamblorul LC-3 pentru stocarea următorului mesaj? (toate numerele din soluţie sunt în sistemul de numeraţie zecimal).

.STRINGZ "Hasta la vista, baby!"

a. 19
b. 20
c. 21
d. 22

4. La adresa 0xA400 se consideră instrucțiunea:

0xA400 LDR R1, R2, 0x10

În plus cunoscând că: (R2) = 0xB000; Mem[xB000] = 0xFFFF;
Mem[0xB010] = 0x000F;
Care este valoarea registrului R1 după execuția instrucțiunii de la adresa 0xA400?

a. 0xA420
b. 0xB000
c. 0xB020
d. 0x000F
e. 0xFFFF

5. Care dintre următoarele instrucțiuni va seta codurile de condiție (n, z, p) bazându-ne doar pe valoarea conținută în registrul R1, fără a modifica conținutul **niciunui** registru? Puteți modifica una dintre instrucțiunile date pentru a mai găsi o soluție ?

a. ADD R2, R1, #0
b. ST R1, LABEL
c. ADD R1, R1, R1
d. AND R1, R1, #0
e. AND R1, R1, # -1

6. Următorul program LC-3 program determină dacă un şir de caractere este sau nu **palindrom**. Un palindrom este un şir de caractere care citit invers (de la dreapta la stânga) este identic cu şirul citit în sensul normal (de la stânga la dreapta) Un astfel de exemplu este şirul "*racecar*". Presupunând că şirul este stocat în memorie începând cu adresa 0x4000, şi se încheie cu valoarea 0x0, similar cu funcția directivei de asamblare .STRINGZ. Dacă şirul aflat în memorie este palindrom, programul se încheie setând valoarea 1 în registrul R5. În caz contrar, programul se încheie, în R5 introducându-se valoarea 0. Se cere să se completeze spațiile goale cu instrucțiunile lipsă pentru ca programul să funcționeze.

```
                .ORIG  0x3000
; Step 1
                LD     R0, PTR
                ADD    R1, R0, #0
```

```
AGAIN   LDR   R2, R1, #0
        BRz   CONT
        ADD   R1, R1, #1
        BRnzp AGAIN

CONT    _____________________
; Step 2
LOOP    LDR   R3, R0, #0

        _____________________
        NOT   R4, R4
        ADD   R4, R4, #1
        ADD   R3, R3, R4
        BRnp  NO
; Step 3
        _____________________

        _____________________
        NOT   R2, R0
        ADD   R2, R2, #1
        ADD   R2, R1, R2
        BRnz  YES

        _____________________
; Step 4
YES     AND   R5, R5, #0
        ADD   R5, R5, #1
        BRnzp DONE
NO      AND   R5, R5, #0
DONE    HALT

PTR     .FILL       x4000
        .END
```


7. Următorul program este asamblat şi executat. Din fericire, nu există erori depistate nici în faza de asamblare nici în cea de execuţie. Ce se afişează pe ecran în urma execuţiei? Presupunem că toţi regiştrii sunt iniţializaţi cu 0 înainte de începerea execuţiei. Se reaminteşte că întreruperea software TRAP 0x22 afişează pe ecran un şir de caractere, a cărui adresă se află în registrul R0, până întâlneşte terminatorul NULL.

```
        .ORIG x3000
        ST R0, x3007
        LEA R0, LABEL
        TRAP x22
        TRAP x25
LABEL
        .STRINGZ "FUNKY"
LABEL2
        .STRINGZ "HELLO WORLD"
        .END
```

8. Completați tabela de simboluri şi asamblați instrucțiunile de la etichetele D, E, şi F în spațiile avute la dispoziție. Se poate considera că la adresa A, înainte de începerea execuției se află un număr pozitiv. Ce execută programul ?

```
        .ORIG 0x3000
        AND R0, R0, #0
D
        LD R1, A
        AND R2, R1, #1
        BRp B
E
        ADD R1, R1, #-1
B
        ADD R0, R0, R1
        ADD R1, R1, #-2
F
        BRp B
        ST R0, C
        TRAP x25
A
        .BLKW 1
C
        .BLKW 1
        .END
```

Tabela de simboluri

Eticheta (*simbol*)	**Valoare**

Adresa instrucțiunii	**Codificarea binară a instrucțiunii**														
D															
E															
F															

9. Ce realizează programul următor ? Care este valoarea stocată la adresa RESULT după încheierea execuției programului ?

```
          .ORIG 0x3000
          LD R2, ZERO
          LD R0, M0
          LD R1, M1
LOOP      BRz     DONE
          ADD R2, R2, R0
          ADD R1, R1, # -1
          BRnzp  DONE
DONE      ST R2, RESULT
          HALT
RESULT    .FILL 0x0000
ZERO      .FILL 0x0000
M0        .FILL 0x0004
M1        .FILL 0x0803
          .END
```

10. Scrieți un program în limbaj de asamblare LC-3 care contorizează numărul de biți de 1 din valoarea stocată în registrul R0 şi depune rezultatul în R1. De exemplu, dacă R0 reține valoarea 0001001101110000, atunci după execuția programului rezultatul stocat în R1 va fi 0000000000000110.

11. Care este scopul directivei de asamblare **.END** ? Prin ce diferă de instrucțiunea **HALT** ?

12. La execuția programului asamblare LC-3, de câte ori se va procesa instrucțiunea de la adresa de memorie etichetată cu LOOP ?

```
          .ORIG 0x3005
          LEA R2, DATA
          LDR R4, R2, #0
LOOP      ADD R4, R4, # -3
          BRzp LOOP
          TRAP 0x25
DATA      .FILL 0x000B
          .END
```

13. Se consideră o secvență de numere întregi nenegative stocată în locații consecutive de memorie (câte un cuvânt – număr întreg – per locație) începând cu adresa 0x4000. Fiecare număr întreg are o valoare între 0 şi 30000 (zecimal). Secvența se încheie când se întâlneşte valoarea -1. Ce realizează programul următor ?

```
            .ORIG 0x3000
            AND R4, R4, #0
            AND R3, R3, #0
            LD R0, NUMBERS
LOOP        LDR R1, R0, #0
            NOT R2, R1
            BRz     DONE
            AND R2, R1, #1
            BRz     L1
            ADD R4, R4, #1
            BRnzp  NEXT
L1          ADD R3, R3, #1
NEXT        ADD R0, R0, #1
            BRnzp  LOOP
DONE        TRAP 0x25
NUMBERS     .FILL 0x4000
            .END
```

14. Se consideră o secvență de numere întregi stocată în locații consecutive de memorie (câte un număr întreg per locație) începând cu adresa 0x4000. Secvența se încheie când se întâlneşte valoarea 0x0000. Ce realizează programul următor ?

```
            .ORIG 0x3000
            LD R0, NUMBERS
            LD R2, MASK
LOOP        LDR R1, R0, #0
            BRz     DONE
            AND R5, R1, R2
            BRz     L1
            BRnzp  NEXT
L1          ADD R1, R1, R1
            STR R1, R0, #0
NEXT        ADD R0, R0, #1
```

```
            BRnzp  LOOP
DONE        TRAP 0x25
NUMBERS     .FILL 0x4000
MASK        .FILL 0x8000
            .END
```

15. Următorul program asamblare LC-3 compară două şiruri de caractere de aceeaşi lungime. Definiția celor două şiruri (stocarea lor în memorie) s-a realizat folosind directiva de asamblare .STRINGZ *şir*. Primul şir începe de la locația FIRST din memorie, iar al doilea de la locația SECOND. Dacă cele două şiruri sunt identice, programul se încheie setând valoarea 0 în R5. Dacă şirurile sunt diferite programul se termină generând valoarea 1 în R5. Completați cele 3 spații libere cu instrucțiunile lipsă astfel încât programul să funcționeze corect.

```
            .ORIG 0x3000
            LD R1, FIRST
            LD R2, SECOND
            AND R0, R0, #0
LOOP        _____________
            LDR R4, R2, #0
            BRz     NEXT
            ADD R1, R1, #1
            ADD R2, R2, #1

            _____________

            _____________

            ADD R3, R3, R4
            BRz     LOOP
            AND R5, R5, #0
            BRnzp  DONE
NEXT        AND R5, R5, #0
            ADD R5, R5, #1
DONE        TRAP 0x25
FIRST       .FILL 0x4000
SECOND      .FILL 0x4100
            .END
```

8. ÎNTRERUPERI SOFTWARE LA NIVEL *LOW*. APELURI DE SUBRUTINE – DIRECTE ŞI INDIRECTE. REVENIRI. SALVAREA ŞI RESTAURAREA REGIŞTRIILOR. STRATEGIILE "*CALLER-SAVE*" RESPECTIV "*CALLEE SAVE*"

8.1. ÎNTRERUPERI. DEFINIȚIE. CLASIFICARE

Prin **întrerupere** se înțelege oprirea programului în curs de execuție şi transferul controlului la o nouă adresă de program [Mus97]. La această adresă se află rutina de tratare a întreruperii, dedicată soluționării cererii de întrerupere. Mecanismul de realizare a transferului este de tipul *apel funcție / revenire*, astfel încât ultima instrucțiune din rutina de tratare trebuie să fie una de revenire (*return*) care să faciliteze întoarcerea în programul principal (cel aflat în execuție în momentul apariției întreruperii), în general[11] pe prima instrucțiune de după cea pe care a apărut întreruperea.

Din punct de vedere al generării lor, întreruperile se clasifică în **hardware** şi **software**. Întreruperile inițiate hardware apar ca răspuns la un semnal extern, fiind de două tipuri: **nemascabile** (sau nedezactivabile) şi **mascabile** (sau dezactivabile). Întreruperile hardware sunt frecvent folosite în calculul de timp real din sistemele multitasking (preluarea controlului de către procesor în anumite situații critice). Întreruperile software sunt de regulă implementate ca şi instrucțiuni în setul de instrucțiuni al fiecărui procesor. Se cunosc două tipuri de întreruperi software:

- **Excepții**, deoarece întreruperea apare numai dacă există o condiție de eroare, care nu permite execuția corespunzătoare a unei instrucțiuni (împărțire cu zero, depăşire de domeniu, etc). Un tip

[11] În general revenirea din rutina de tratare a întreruperii se face pe instrucțiunea imediat următoare celei care a cauzat întreruperea. Pot însă apărea instrucțiuni de genul *Load Adresă* care să cauzeze o excepție de tip *Page Fault*. În această situație va fi tratată excepția după care se va relua execuția programului cu aceeaşi instrucțiune de acces la memorie (*Load Adresa*).

special de excepție îl reprezintă *excepția de depanare*, care permite execuția unui program instrucțiune cu instrucțiune („*pas cu pas"*).

- **Întreruperi generate la fiecare execuție a instrucțiunii TRAP *n*** (unde $n \in (0 \div 255)$).

Din punct de vedere al sincronizării cu ceasul procesorului, întreruperile *software* sunt evenimente **sincrone**, reprezentând răspunsuri ale procesorului la anumite evenimente detectate în timpul execuției unei instrucțiuni, în timp ce întreruperile *hardware* sunt evenimente **asincrone**, fiind generate de dispozitive externe. Întreruperile software sunt întotdeauna reproductibile prin reexecuția programului în aceleaşi condiții de intrare, în timp ce întreruperile hardware sunt de obicei independente de execuția procesului curent.

8.2. ÎNTRERUPERI SOFTWARE LA NIVEL *LOW* – INSTRUCȚIUNILE *TRAP*

La nivelul unui sistem de calcul există câteva instrucțiuni prin care un anumit program poate afecta comportamentul altuia (numite *privilegiate*) şi care sunt cel mai bine executate de către un program *supervizor* (sistemul de operare) şi nu de către programele *utilizator*.

- Instrucțiuni de **I**ntrare / I**e**şire care necesită cunoştințe specifice de utilizare şi protecție a regiştrilor de interfață cu dispozitivele periferice (KBDR, KBSR – tastatura şi CRTSR, CRTDR – monitorul, etc). Întrucât dispozitivele periferice sunt partajate între mai multe programe şi mai mulți utilizatori, o greşală dintr-un program poate afecta mai mulți utilizatori.
- Instrucțiuni pentru resetarea ceasului sistemului (la acționarea butonului *Reset* sau la oprirea sistemului).
- Instrucțiuni care marchează încheierea aplicației utilizator şi cedarea controlului sistemului de operare.
- Încărcarea regiştrilor aferenți dispozitivelor periferice şi mapați în memoria sistemului de calcul (*memory-mapped registers*[12])

[12] Majoritatea proiectanților de sisteme de calcul preferă să nu specifice un set de instrucțiuni suplimentare pentru tratarea interacțiunii dintre procesor şi dispozitivele periferice. Aceştia folosesc aceleaşi instrucțiuni de transfer date utilizate la citirea / scrierea datelor din / în memorie în / din regiştrii procesorului (load / store). Astfel, o instrucțiune Load a cărui adresă sursă este specificată printr-un registru aferent unui dispozitiv periferic

Unitatea centrală de procesare (CPU) poate fi proiectată să lucreze în două moduri:

- Mod utilizator
- Mod privilegiat (numit şi supervizor, nucleu, monitor).

Doar programele supervizor pot executa instrucțiuni privilegiate. În plus, nu toți programatorii cunosc sau își doresc să afle aceste informații la un nivel foarte ridicat de detaliu.

8.2.1. RUTINELE DE TRATARE AFERENTE APELURILOR SISTEM LA LC-3 ISA

Întreruperile software la nivel *low* numite **apeluri sistem** reprezintă un mic set de servicii ale sistemului de operare prin instrucțiuni (*syscall* – la procesorul MIPS, întreruperi software *int* 21h la Intel, TRAP la arhitectura LC-3) care au rolul de a efectua în siguranță operații privilegiate. Pentru a apela un serviciu, trebuie încărcat codul apelului sistem într-unul din regiştrii arhitecturali (la MIPS în $2 ($v0), la LC-3 în R0, la Intel în AH). Protocolul de efectuare a unui apel sistem se realizează în etapele:

1. Utilizatorul invocă un apel sistem.
2. Este predat controlul sistemului de operare care determină execuția secvenței de cod aferentă rutinei de tratare a întreruperii respective.
3. Returnează controlul programului utilizator.

În cazul arhitecturii LC-3 acest protocol se numeşte *mecanismul TRAP*. Instrucțiunile TRAP izolează secțiunile de cod critice de utilizator. Mecanismul TRAP se bazează pe următoarele componente:

1. **Mulțimea rutinelor de serviciu** (de tratare a diferitelor întreruperi – rutinele TRAP)

---

de intrare în sistem, se numeşte instrucțiune de intrare. Similar, o instrucțiune Store a cărui adresă destinație este specificată printr-un registru aferent unui dispozitiv periferic de ieşire din sistem, se numeşte instrucțiune de ieşire. Întrucât sunt folosite aceleași tipuri de instrucțiuni de acces la memorie şi pentru periferice, regiştrii dispozitivelor de intrare / ieşire trebuie identificați în mod unic în același fel cum sunt identificate locațiile de memorie. Astfel, fiecărui registru aferent unui dispozitiv periferic îi este asignată o adresă în spațiul de memorie descris de ISA-ul fiecărui procesor. Practic **regiştrii de intrare / ieşire sunt mapați spre o zonă de adrese specifică dispozitivelor periferice şi nu memoriei de date**. Exemple de familii de procesoare care folosesc regiştrii aferenți dispozitivelor periferice mapați în memorie în combinație cu instrucțiuni load / store sunt MIPS, SPARC şi Motorola, iar Intel foloseşte instrucțiuni speciale de intrare / ieşire incluse în ISA (IN / OUT) pentru citirea sau scrierea datelor din / în regiştrii dispozitivelor periferice.

- Sunt parte integrantă a sistemului de operare; adresele de început a fiecărei rutine sunt arbitrar aleseîn spaţiul de memorie 0x0000÷0x2FFF sau 0xCFFF÷0xFFFF.
- Pot fi implementate până la 256 de rutine la arhitectura LC-3 ISA.
- Rutinele de serviciu asigură trei funcţii speciale:
  - Izolează programatorii de detalii specifice sistemului de operare.
  - Permite scrierea codului frecvent utilizat o singură dată şi apelarea de câte ori este nevoie.
  - Protejează resursele sistemului de programatorii neîndemânatici.

2. **Tabela adreselor de start**
   - Adresele de start sunt memorate într-o tabelă de 256 de intrări începând cu adresa 0x0000 şi până la adresa 0x00FF, numită **tabela vectorilor de întrerupere TRAP** sau blocul de control sistem în cazul altor arhitecturi. Deşi în versiunea actuală a arhitecturii LC-3 ISA sunt implementate doar 5 apeluri sistem, orice tentativă de folosire (scriere / citire) a zonei de adrese 0x0000÷0x001F va genera o excepţie.
3. **Instrucţiunile TRAP**
   - Sunt folosite în programele utilizator pentru a transfera controlul sistemului de operare. Codifică în corpul instrucţiunii un vector de întrerupere pe 8 biţi (codul apelului sistem) care reprezintă un index în tabela cu adresele de start a rutinelor de serviciu. Informaţia de la locaţia specificată reprezintă adresa primei instrucţiuni din rutina de tratare a întreruperii, valoare care se încarcă în registrul PC.
4. **Instrucţiunea RET**
   - Trebuie să fie prezentă în fiecare rutină de tratare a întreruperii, având rolul de a transfera controlul de la nivelul sistemului de operare la nivelul programului utilizator (în cazul procesorului INTEL se numeşte *IRET*, iar la MIPS se numeşte *rfe – return from exception*). Revenirea se face pe instrucţiunea imediat următoare instrucţiunii TRAP.

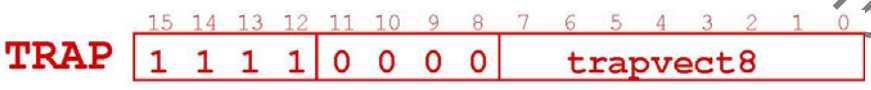


Figura 8.1. Formatul instrucţiunii TRAP

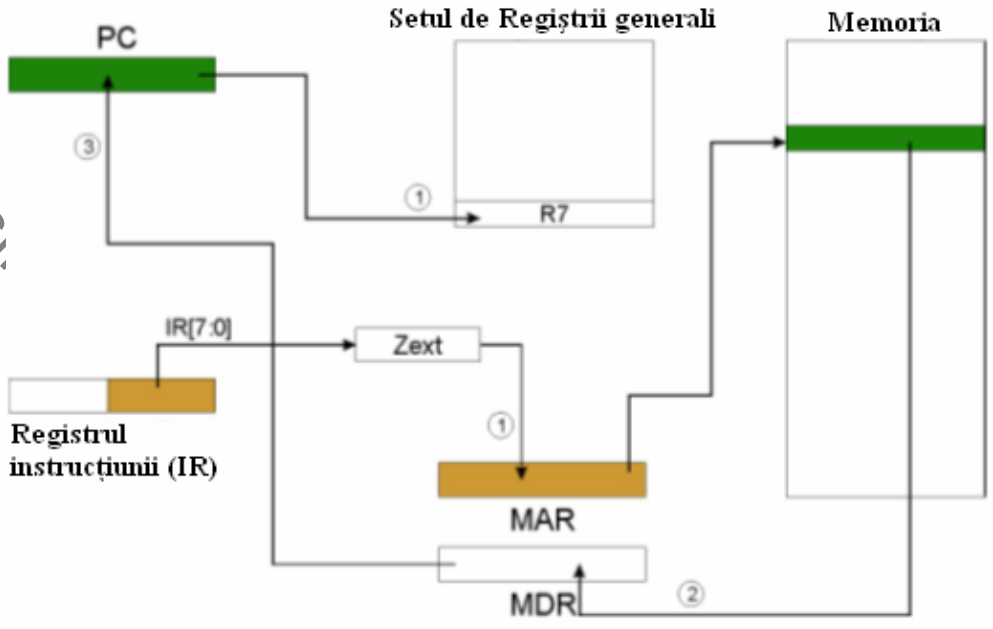


Figura 8.2. Modul de lucru al instrucțiunii TRAP

În figura 8.1 este prezentat formatul instrucțiunii TRAP iar figura 8.2 ilustrează modul de lucru al acesteia. Primii 4 biți ai instrucțiunii sunt 1 urmați de 4 biți de 0 iar ultimii 8 biți reprezintă codul apelului. În registrul R7 se încarcă valoarea PC-ului (care în momentul execuției instrucțiunii TRAP pointează deja la instrucțiunea următoare din program, PC-ul fiind incrementat în timpul fazei FETCH) – vezi operația **1** din figura 8.2. Întrucât adresele de început ale rutinelor de tratare sunt stocate în memorie de la adresa 0x0000 la adresa 0x00FF rezultă că pentru selecția adresei de început a rutinei corespunzătoare, în registrul MAR, cei mai semnificativi 8 biți ai codului de apel (din registrul IR aferent instrucțiunii TRAP) se extind cu 0 pe 8 biți – vezi operația **1** din figura 8.2. În PC se încarcă adresa primei instrucțiuni din rutina de tratare (valoare citită din memorie de la adresa indicată de MAR şi depusă apoi în MDR– vezi operația **2** din figura 8.2), în acest fel fiind predat controlul sistemului de operare – vezi operația **3** din figura 8.2. Trebuie specificat că în cadrul arhitecturii LC-3 ISA, rolul registrului **R7 este de adresă de revenire în programul apelant** iar **R6 are rolul de indicator de stivă** (pointer la ultima locație – cea mai recent – ocupată în stivă). Toate arhitecturile RISC dețin astfel de regiştrii (la MIPS – $ra şi $sp).

Instrucțiunea RET trebuie să fie practic ultima instrucțiune din rutina de tratare a întreruperii (sau a oricărei rutine). Rolul său este de a transfera

conţinutul registrului R7 în registrul PC cedând astfel controlul înapoi programului apelant (utilizator) pe instrucţiunea imediat următoare instrucţiunii TRAP (doar dacă nu cumva a fost alterată în interiorul rutinei). Formatul instrucţiunii RET este cel din figura 8.3 iar modul său de lucru este ilustrat în figura 8.4.

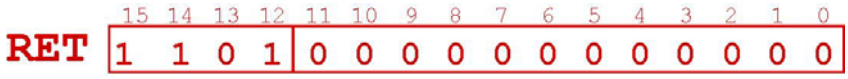


Figura 8.3. Formatul instrucţiunii RET

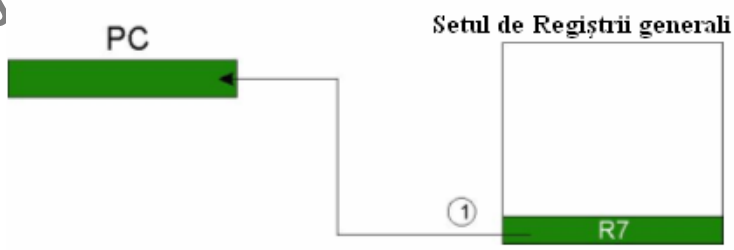


Figura 8.4. Modul de lucru al instrucţiunii RET

Vector	Simbol	Operaţii efectuate
x20	**GETC**	• Citeşte un singur caracter de la tastatură fără ecou. • Scrie codul ASCII al caracterului citit în registrul R0[7:0], biţii R0[15:8] setându-i pe 0.
x21	**OUT**	Afişează pe monitor caracterul al cărui cod ASCII se regăseşte în registrul R0[7:0].
x22	**PUTS**	Afişează pe consolă şirul de caractere al cărui adresă de început se află în registrul R0.
x23	**IN**	• Afişează un mesaj pe consolă şi citeşte fără ecou un caracter de la tastatură. • Înscrie codul ASCII al caracterului citit în R0[7:0] şi resetează biţii R0[15:8].
x25	**HALT**	Afişează un mesaj de atenţionare pe ecran şi întrerupe execuţia programului.

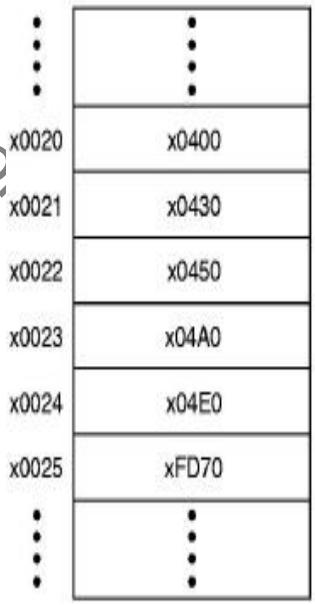


Figura 8.5. Semantica instrucţiunilor TRAP. Adresele de start aferente rutinelor de tratare ale întreruperii

Se consideră următorul exemplu care foloseşte apelurile sistem de *citire / scriere caracter* şi *încheiere program*. Problema care se rezolvă *citeşte un caracter de la tastatură atâta timp cât este diferit de cifra 7 şi la codul său ASCII se adaugă 0x20* (*dacă s-ar tasta doar litere mari ale alfabetului englez atunci programul le transformă în litere mici*).

```
            .ORIG x3000
            LD     R2, TERM;    Încarcă inversul codului ASCII al
                                cifrei '7' (-55)
            LD     R3, ASCII;   Încarcă diferența de cod ASCII dintre
                                litere mici şi mari
AGAIN:      TRAP   x23;         Se citeşte un caracter de la tastatură
            ADD R1, R2, R0;     Se verifică dacă s-a tastat '7' în caz
                                afirmativ încheindu-se execuția
                                programului
            BRz  EXIT   ;       S-a tastat '7' salt la apelul sistem de
                                ieşire din program
            ADD R0, R0, R3;     Se adaugă diferența de 0x20 la codul
                                ASCII al caracterului introdus de la
                                tastatură
            TRAP   x21      ;   Se afişează pe monitor noul caracter
                                obținut
            BRnzp  AGAIN;       Se reia execuția până se va tasta '7' de
                                la tastatură
TERM:       .FILL  xFFC9    ;   -'7'
ASCII:      .FILL  x0020    ;   Diferența dintre litere mici şi mari
EXIT:       TRAP   x25      ;   Se încheie execuția programului
            .END
```

Obs: Codul ASCII pentru cifra '7' este 0x37=**55** în zecimal. Trebuie specificat că cifra '7' a fost aleasă doar pentru test, putând fi aleasă oricare altă valoare (caracter) de test.

Secvența următoare de cod reprezintă conținutul rutinei de tratare aferentă apelului sistem de afişare caracter – ***Output Service Routine* (TRAP 0x21).**

```
                .ORIG x0430  ;          Adresa de start a rutinei de tratare a
                                        apelului TRAP 0x21
                ST R7, SaveR7;          Salvează conţinutul regiştrilor R7 şi
                                        R1 – posibil a fi modificaţi pe
                                        parcursul rutinei de tratare
                ST      R1, SaveR1
                ; ----- Scrie caracter
TryWrite        LDI R1, CRTSR;          Citeşte starea monitorului (registrul
                                        CRTSR). Dacă bitul cel mai
                                        semnificativ este 1 atunci se înscrie
                                        codul ASCII al caracterului de afişat în
                                        registrul de date al monitorului.
                BRzp  TryWrite;         Se testează bitul cel mai semnificativ
                                        al registrului de stare
WriteIt         STI R0, CRTDR;          Se înscrie codul ASCII al caracterului
                                        de afişat în registrul de date al
                                        monitorului (registrul CRTDR)
                ; ----- Revenire din apelul sistem de afişare TRAP 0x21
Return          LD R1, SaveR1;          Restaurează conţinutul regiştrilor R7 şi
                                        R1 care au fost modificaţi pe parcursul
                                        rutinei de tratare
                LD      R7, SaveR7
                RET                     ;Transferă controlul înapoi utilizatorului
CRTSR           .FILL  xF3FC
CRTDR           .FILL  xF3FF
SaveR1          .FILL  0
SaveR7          .FILL  0
                .END
```

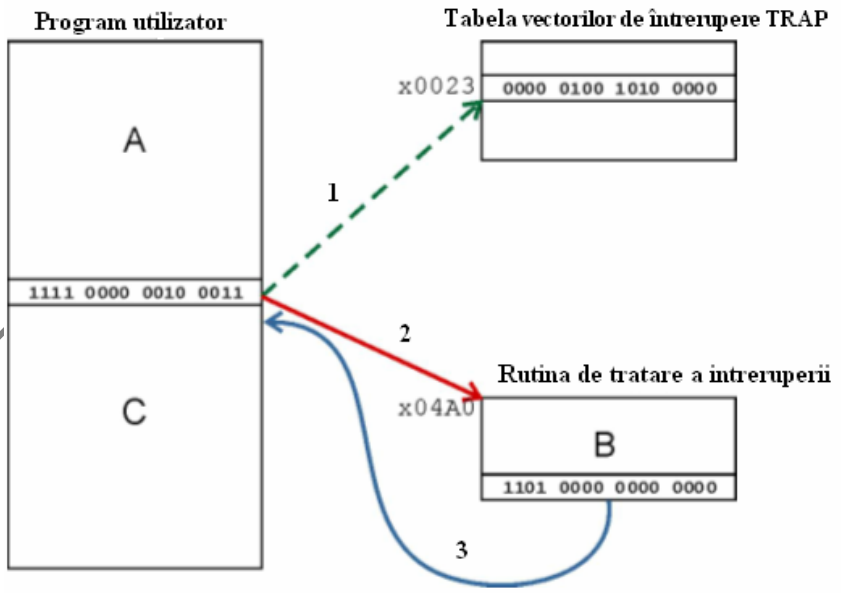


Figura 8.6. Etapele parcurse în urma apariției unui apel sistem de citire caracter în programul utilizator

Secvența următoare de cod reprezintă conținutul rutinei de tratare aferentă apelului sistem de citire caracter cu ecou – ***Character Input Service Routine* (TRAP 0x23).** Figura 8.6 ilustrează etapele majore parcurse din momentul apariției apelului şi până la revenirea din rutina de tratare. Pe scurt, în rutina de tratare se desfăşoară următoarele subetape: se afişează caracterul special *newline* (deci implicit sunt incluse instrucțiunile de afişare caracter), urmat de afişarea unui mesaj care solicită introducerea unui caracter de la tastatură (sunt incluse instrucțiunile de afişare şir de caractere). Urmează citirea propriu-zisă de la tastatură iar apoi afişarea ecoului şi în final trecerea pe o nouă linie.

```
                .ORIG x04A0
; ----- Salvează conținutul regiştrilor care vor fi alterați la nivelul subrutinei
START   ST    R7, SaveR7    ; Salvează adresa de revenire în
                              programul apelant (R7)
        ST    R1, SaveR1    ; Salvează valorile regiştrilor necesari
                              în rutina: R1, R2 şi R3
        ST    R2, SaveR2
        ST    R3, SaveR3    ; trebuie restaurați înaintea execuției
                              instrucțiunii RET
```

```
; ----- Scrie caracter special newline
                LD      R2, Newline   ; Se încarcă în R2 codul ASCII al
                                        caracterului special newline
L1              LDI     R3, CRTSR     ; Verifică starea perifericului – dacă
                                        permite scrierea în registrul de date al
                                        monitorului CRTDR este disponibil?
                BRzp    L1            ; Registrul CRTDR este disponibil
                                        dacă bitul cel mai semnificativ al
                                        registrului de stare (cel de semn –
                                        CRTSR15) este 1. Dacă nu este încă
                                        disponibil pentru afişare atunci se va
                                        continua cu citirea stării acestuia.
                STI     R2, CRTDR     ; Este permisă afişarea caracterului
                                        Newline la consolă
; ----- Afişează pe ecran mesajul de atenţionare pentru citirea unui caracter
        de la tastatură
                LEA     R1, Prompt    ; Prompt reprezintă adresa de start a
                                        mesajului de atenţionare
Loop            LDR     R0, R1, #0    ; Încarcă în R0 caracter după caracter
                                        din mesajul de afişat
                BRz     Input         ; Dacă s-a ajuns la sfârşitul mesajului
                                        trece la citirea caracterului de la
                                        tastatură
L2              LDI     R3, CRTSR     ; Verifică starea monitorului
                BRzp    L2
                STI     R0, CRTDR     ; Periferic disponibil ⇒ afişează
                                        caracterul curent al mesajului
                ADD     R1, R1, #1    ; Trece la următorul caracter din mesaj
                BRnzp   Loop          ; Repetă afişarea până la sfârşitul
                                        şirului
; ----- Citire caracter de la tastatură
Input           LDI     R3, KBSR      ; Verifică starea tastaturii – s-a apăsat
                                        vreo tastă?
                BRzp    Input         ; Registrul KBDR este disponibil dacă
                                        bitul cel mai semnificativ al registrului
                                        de stare (cel de semn – KBSR15) este
                                        1. Dacă nu este încă disponibil pentru
```

```
                                    afişare atunci se va continua cu citirea
                                    stării acestuia.
                LDI     R0, KBDR    ; Preia caracterul apăsat de la tastatură
                                      în registrul R0
; ----- Afişează pe ecran ecoul caracterului citit
L3              LDI     R3, CRTSR
                BRzp    L3
                STI     R0, CRTDR   ; Afişează caracterul citit de la
                                      tastatură pe monitor
; ----- Scrie caracter special newline – trece la o nouă linie
L4              LDI     R3, CRTSR
                BRzp    L4
                STI     R2, CRTDR   ; Mută cursorul pe ecran la o linie
                                      nouă
; ----- Reface conținutul regiștrilor care au fost alterați la nivelul subrutinei
                LD      R1, SaveR1  ; Preluarea lui R1, R2 şi R3 de la
                                      adresele unde au fost salvați
                LD      R2, SaveR2
                LD      R3, SaveR3
                LD      R7, SaveR7  ; Restaurarea lui R7 la valoarea avută
                                      la intrarea în rutina de serviciu
                RET                 ; Revenirea din rutina de serviciu pe
                                      prima instrucțiune în programul
                                      apelant ulterioară TRAP 0x23
                                      (echivalentă cu: JMP R7)
; ----- Spațiul aferent datelor necesare rutinei de serviciu
SaveR7          .FILL 0x0000
SaveR1          .FILL 0x0000
SaveR2          .FILL 0x0000
SaveR3          .FILL 0x0000
CRTSR           .FILL 0xF3FC        ; adresa de memorie a registrului de
                                      stare al monitorului
CRTDR           .FILL 0xF3FF        ; adresa de memorie a registrului de
                                      date al monitorului
KBSR            .FILL 0xF400        ; adresa de memorie a registrului de
                                      stare al tastaturii
KBDR            .FILL 0xF401        ; adresa de memorie a registrului de
```

```
                                    date al tastaturii
Newline     .FILL 0x000A            ; codul ASCII al caracterului special
                                    newline
Prompt      .STRINGZ "Input a character>"      ; mesaj de atenţionare
            .END                    ; încheierea secvenţei de cod
```

Se reaminteşte că oprirea sistemului de calcul presupune oprirea unităţii de control, deci anularea semnalului de tact care reprezintă „*pulsul*" sistemului (vezi capitolul 5, figura 5.8). Arhitectura setului de instrucţiuni aferentă primelor sisteme de calcul conţineau o instrucţiune specială (HALT) pentru oprirea sistemului. Ulterior, proiectanţii acestora au realizat cât de rar este folosită instrucţiunea HALT şi totodată, faptul că se iroseşte un cod de operaţie pentru stocarea ei, care uneori poate fi necesar pentru definirea altor instrucţiuni (operaţii) cu caracter mult mai frecvent. La ora actuală, sistemele de calcul folosesc apeluri (rutine) sistem pentru anularea semnalului de tact şi implicit oprirea unităţii de control. În cadrul LC-3 ISA, prin întreruperea TRAP 0x25 se urmăreşte practic oprirea ceasului sistemului, însă, aflându-ne într-un simulator (aplicaţie software) şi nu pe un calculator gazdă, practic se opreşte funcţionarea simulatorului LC-3 (a aplicaţiei care a rulat pe acesta) şi se predă controlul sistemului de operare care rulează şi gestionează procesele aferente calculatorului gazdă.

Secvenţa următoare de cod reprezintă conţinutul rutinei de tratare aferentă apelului sistem de încheiere program şi transfer control sistemului de operare – ***Halt Service Routine*** **(TRAP 0x25).** Pe scurt, în rutina de tratare se desfăşoară următoarele subetape: se afişează caracterul special *newline* (folosindu-se apelul sistem TRAP 0x21), urmat de afişarea unui mesaj care atenţionează asupra încheierii aplicaţiei utilizator şi cedarea controlului sistemului de operare (se foloseşte apelul sistem TRAP 0x22 pentru afişarea şirului de caractere). Urmează apoi resetarea bitului cel mai semnificativ al registrului de control al maşinii (MCR_{15}) localizat în memoria LC-3 la adresa 0xFFFF.

```
            .ORIG 0xFD70        ; Adresa de început a rutinei
            ST    R7, SaveR7    ; Salvează adresa de revenire în
                                programul apelant (R7)
            ST    R1, SaveR1     ; R1 este folosit să stocheze temporar
                                 registrul de control al maşinii (MCR)
            ST    R0, SaveR0    ; R0 este folosit ca temporar în
                                operaţiile din rutina de serviciu
```

; ----- *Afişează pe ecran mesajul de atenţionare pe linie nouă*

```
        LD      R0, ASCIINewLine
        TRAP    0x21
        LEA     R0, Message
        TRAP    0x22
        LD      R0, ASCIINewLine
        TRAP    0x21
```

; ----- Resetarea bitului cel mai semnificativ al registrului de control al maşinii (MCR_{15}) pentru *oprirea ceasului sistemului*

```
        LDI     R1, MCR     ; Încarcă conținutul registrului de
                            control al maşinii în R1. Registrul –
                            fiind reprezentat de o locație de
                            memorie – este necesar accesul
                            indirect.
        LD      R0, MASK    ; R0 = 0x7FFF, reprezintă masca
                            pentru ştergerea bitului 15 al MCR
        AND     R0, R1, R0  ; Ceilalți biți ai registrului MCR rămân
                              nemodificați
        STI     R0, MCR     ; Scrie în memorie valoarea modificată
                              a MCR
```

; ----- Revenirea din rutina sistem dedicată opriri ceasului sistemului

```
        LD      R1, SaveR1  ; Restaurarea regiştrilor alterați în
                            rutină. Preluarea lui R0 şi R1 de la
                            adresele unde au fost salvați
        LD      R0, SaveR0
        LD      R7, SaveR7  ; Restaurarea lui R7 la valoarea avută
                            la intrarea în rutina de serviciu
        RET                 ; Revenirea din rutina de serviciu în
                            sistemul de operare (procesul părinte
                            care a lansat în execuție simulatorul
                            LC-3).
```

; ----- *Spațiul aferent datelor necesare rutinei de serviciu*

```
ASCIINewLine    .FILL   0x000A
SaveR0          .FILL   0x00000
SaveR1          .FILL   0x00000
SaveR7          .FILL   0x00000
Message         .STRINGZ    "Halting the machine."
```

```
MCR             .FILL   0xFFFF      ; Adresa de memorie a
                                      registrului MCR
MASK            .FILL   0x7FFF      ; Masca necesară resetării
                                      bitului 15 al MCR
                .END                ; încheierea secvenţei de cod
```


8.2.2. SALVAREA ŞI RESTAUREA REGIŞTRILOR ÎN CAZUL RUTINELOR DE SERVICIU

Conţinutul unui registru trebuie salvat dacă registrul va fi modificat de către rutina de serviciu iar valoarea acestuia va fi necesară şi după încheierea rutinei. Se pune întrebarea cine salvează valoarea registrului:

- **Apelantul** rutinei de tratare (programul utilizator) care ştie ce regiştrii sunt necesari după încheierea rutinei, dar nu ştie ce regiştrii sunt modificaţi în rutină ? Salvările efectuate la nivelul apelantului poartă numele de „**caller-save**” şi presupune stocarea temporară în memorie a conţinutului regiştrilor care sunt distruşi de către programul apelant sau în rutinele apelate (dacă este vizibil sau apriori cunoscut ceea ce se modifică în rutină) şi valorile acestora sunt necesare ulterior. În cazul arhitecturii LC-3 ISA, strategia „caller-save” impune salvarea lui R7 înaintea oricărei instrucţiuni TRAP şi de asemenea, salvarea lui R0 înaintea apelului TRAP 0x23 (deoarece rezultatul acestuia presupune stocarea codului ASCII al caracterului citit de la tastatură în R0). O altă posibilitate, realistă în cazul arhitecturilor cu număr ridicat de regiştri generali ai procesorului, presupune evitarea folosirii acestor regiştrii cu funcţii speciale (adresă de revenire, indicator de stivă, registrul care păstrează valoarea returnată din (sub)rutine) în operaţii uzuale (aritmetico-logice, etc).

sau

- **Rutina apelată** (în acest caz cea de tratare a întreruperii) ? Avantajul acestei variante este că, rutina cunoaşte regiştrii pe care îi va modifica, dar nu cunoaşte ce regiştrii vor fi necesari ulterior după execuţia rutinei. În acest caz, strategia de salvare a regiştrilor se numeşte „**callee-save**”. Înainte de intrarea propriu-zisă în (sub)rutină sunt salvaţi toţi regiştrii care vor fi alteraţi, exceptându-i pe cei care, chiar dacă sunt alteraţi, nu sunt necesari la nivelul programului apelant (totuşi, acest lucru nu este întotdeauna vizibil de la nivelul (sub)rutinei apelate). Înainte de revenirea din (sub)rutină, regiştrii salvaţi iniţial sunt restauraţi la valorile anterioare apelului. Opţiunea „callee-save”

este utilizată în mod frecvent şi în cazul apelurilor de subrutine (rutine definite de programator / utilizator) nu doar în situaţia rutinelor de tratare a întreruperilor.

Salvarea şi restaurarea regiştrilor stau la baza unui principiu fundamental al programării ce poartă numele de ***reguli de scop*** în cadrul limbajelor de nivel înalt. Regulile de scop informează despre faptul că o entitate de program (variabilă, parametru al funcţiei etc.) este „vizibil" sau accesibil într-un anumit loc. Astfel, locul în care o entitate este vizibilă este referit ca **scopul** entităţii.

În continuare se prezintă un exemplu care dovedeşte necesitatea salvării registrului R7 (adresa de revenire) de către programul apelant înaintea execuţiei unui apel sistem (*TRAP 0x23* în acest caz). Programul urmăreşte stocarea a 10 caractere (numerice) citite de la tastatură sub formă de cifre de la 0 la 9. Din păcate nu se întâmplă acest lucru, mai mult, programul intră în buclă infinită. Greşeala este provocată de faptul (ascuns la prima vedere) că apelul sistem TRAP 0x23 modifică valoarea lui R7 cu adresa instrucţiunii ADD R0, R0, R6 (0x3004). La revenirea din apel R7 va avea această valoare, pozitivă, se va executa saltul la AGAIN şi se reexecută apelul sistem, şamd, intrându-se astfel în buclă infinită.

Salvarea adresei de revenire în programul apelant trebuie făcută în cazul oricărui apel imbricat de rutină (incluzându-l şi pe cel recursiv), altfel se pierde legătura spre programul apelant şi revenirea se face la infinit pe aceeaşi instrucţiune într-una din rutinele apelate.

```
        .ORIG x3000
        LEA R3, Binary;     În R3 se încarcă adresa de început a
                            tabloului de numere
        LD R6, ASCII;       R6 conţine valoarea -48 care trebuie
                            scăzută din fiecare caracter (cifră) citit
                            de la tastatură pentru determinarea
                            valorii numerice a cifrei
        LD   R7, COUNT;     Iniţializarea lui R7 cu 10
AGAIN   TRAP x23;           Citeşte un caracter de la tastatură (în
                            R0 se află codul său ASCII)
        ADD  R0, R0, R6;    Transformă caracterul în număr
        STR  R0, R3, #0;    Stochează numărul în tablou
        ADD  R3, R3, #1;    Incrementează poziţia în tablou
        ADD R7, R7, -1;     Decrementează contorul de elemente
                            ce mai trebuie citite de la tastatură
        BRp  AGAIN;         Mai sunt elemente de citit ?
```

```
            BRnzp NEXT;     S-au citit cele 10 elemente, salt la
                            încheierea programului
ASCII       .FILL   xFFD0;  Valoarea ASCII = -48
COUNT       .FILL   #10
Binary      .BLKW #10
NEXT        HALT
            .END
```

Ce modificări trebuie aduse acestei secvenţe pentru ca la locaţia Binary să fie stocate numerele (şi nu codurile lor ASCII) citite de la tastatură ?

8.3. SUBRUTINE

Subrutinele sunt secţiuni de cod (fragmente de program) definite de programator / utilizator şi care:

- Sunt stocate în spaţiul de memorie utilizator
- Realizează o sarcină bine precizată (execută operaţii bine definite de către programator)
- Este invocată (apelată) de către un alt program utilizator (sau subrutină) – program apelant
- La încheierea ultimei instrucţiuni din subrutină se predă controlul programului apelant pe instrucţiunea succesoare apelului.

Subrutinele diferă de rutinele de tratare a întreruperilor prin faptul că nu sunt parte componentă a sistemului de operare, nu necesită privilegii speciale şi nu fac uz de resurse hardware protejate.

Există câteva motive în favoarea utilizării subrutinelor:

- Reutilizarea, eficientă din punct de vedere al timpului de execuţie, a codului, fără a fi necesară rescrierea lui. O consecinţă a acestui fapt îl reprezintă reducerea numărului de erori care pot apărea la scrierea unui program.
- Elaborarea algoritmilor prin descompunerea unei probleme în *subprobleme* mai simple şi distribuirea sarcinilor între mai mulţi programatori – fiecare realizează un anumit task, independent de ceilalţi. În acest fel creşte lizibilitatea codului, el putând fi mai uşor de urmărit, depanat şi corectat.

- Cele mai frecvent folosite funcţii (subrutine) matematice, grafice, sau cu diferite caracteristici, sunt implementate şi furnizate de producătorii de sisteme software, sub forma funcţiilor de bibliotecă. Se impune însă respectarea câtorva cerinţe:
- Transmiterea parametrilor subrutinei apelate şi returnarea rezultatului de către aceasta subrutinei apelante (eventual programului principal) se face prin regiştrii sau memorie. De exemplu, în cazul procesorului MIPS, în regiştrii $a0, $a1, $a2, $a3 sunt transmişi parametrii subrutinei apelate, iar în caz că sunt mai mulţi de 4 aceştia se depun pe stivă. Rezultatul returnat de subrutine este transmis prin intermediul regiştrilor $v0 şi $v1 dacă este necesar.
- Subrutinele trebuie să poată fi apelate din orice punct al programului (sau subrutine componente), revenirea făcându-se pe instrucţiunea imediat următoare apelului.

8.3.1. MECANISMUL DE APEL SUBRUTINE ŞI REVENIRE

Figura 8.7. ilustrează execuţia unui program sursă conţinând mai multe secţiuni de cod independente – secvenţele contigue din punct de vedere al execuţiei X, A, Y, Z şi W. Trebuie menţionat că secvenţa de cod A se repetă de câteva ori astfel încât ar fi recomandabilă stocarea codului într-o subrutină, scrisă o singură dată şi apelată repetat.

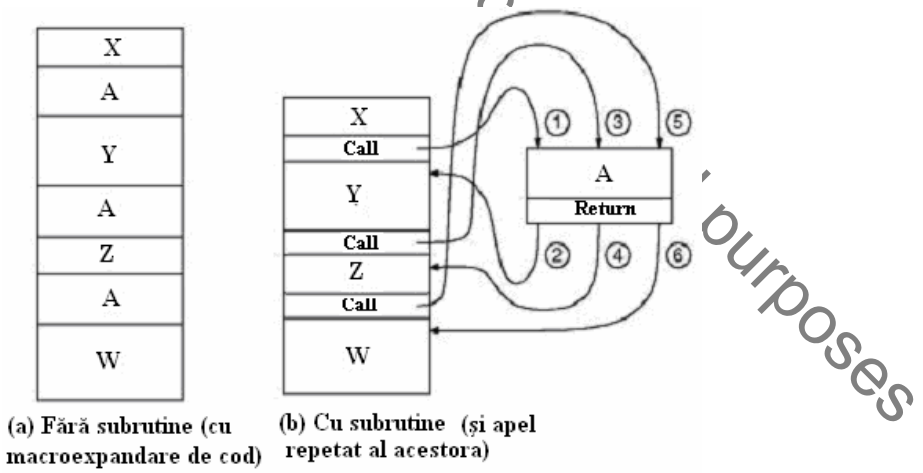


Figura 8.7. Execuţia unui program sursă: (a) fără subrutine prin macroexpandarea[13] codului, (b) folosind subrutine şi apelul repetat al acestora

[13] Repetarea (scrierea conţinutului rutinei sau funcţiei în locul apelului său).

Scopul principal al instrucțiunilor **JSR/JMP** este de a efectua un salt necondiționat la o adresă. Deosebirea dintre ele constă în faptul că, instrucțiunea JSR reprezintă mai mult decât un salt necondiționat, este chiar un apel de subrutină, întrucât presupune suplimentar şi salvarea în R7 a adresei de revenire (PC-ul următoarei instrucțiuni) în programul apelant – operație numită **linking** (*păstrarea legăturii cu programul apelant*) cu implicații importante şi asupra stivei de date aferentă funcțiilor (vezi capitolul 10). Diferențierea efectivă între cele două instrucțiuni se face prin bitul *L* (IR[11]) din formatul instrucțiunii (vezi figura 8.8). Astfel dacă L=0 instrucțiunea este simplu salt necondiționat (se rămâne în programul apelant).

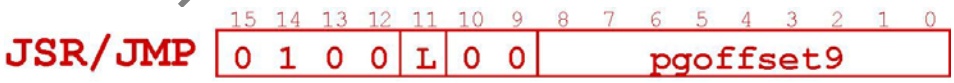


Figura 8.8. Formatul instrucțiunilor JSR/JMP

În figura 8.8 este prezentat formatul instrucțiunii JSR / JMP iar figura 8.9 ilustrează modul de lucru al acesteia. Primii 4 biți ai instrucțiunii sunt 0100 urmați de un bit L (L=1 specifică automatului cu stări finite din unitatea de control aferentă arhitecturii LC-3 că instrucțiunea este un apel de subrutină şi nu un salt necondiționat) şi 2 biți de 0. Ultimii 9 biți reprezintă deplasamentul (adresa relativă pozitivă sau negativă) la care se află subrutina față de adresa următoarei instrucțiuni. În registrul R7 se încarcă valoarea PC-ului (care în momentul execuției instrucțiunii JSR pointează deja la instrucțiunea următoare din programul apelant, PC-ul fiind incrementat în timpul fazei FETCH) – vezi operația **1** din figura 8.9. În PC se încarcă adresa primei instrucțiuni din subrutina apelată, predându-se controlul acesteia – vezi operația 2 din figura 8.9. Adresa primei instrucțiuni din subrutina apelată se obține prin adăugarea la valoarea PC-ului curent a extensiei de semn a câmpului offset din formatul instrucțiunii de apel (PC ← (PC) + Sext(IR[8:0])).

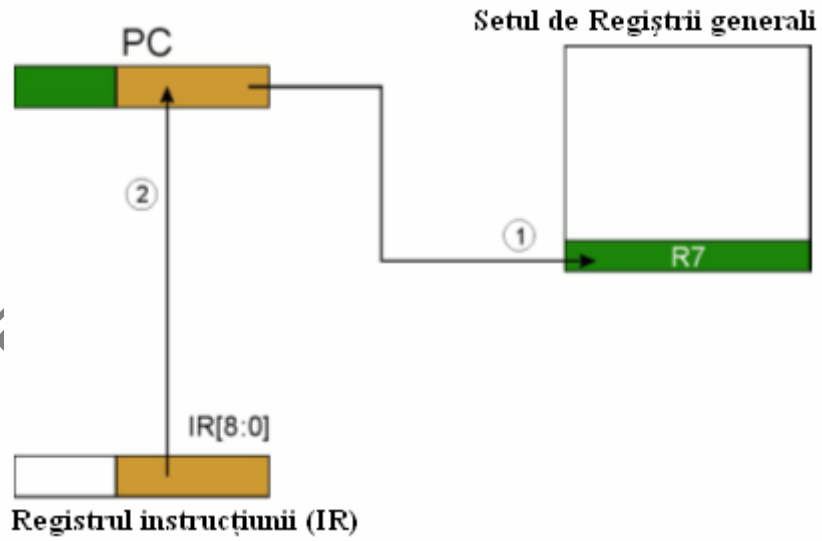


Figura 8.9. Modul de lucru al instrucţiunii **JSR**

În figura 8.10 este prezentat formatul instrucţiunilor JSRR / JMPR iar figura 8.11 ilustrează modul de lucru al acesteia. Semantica instrucţiunilor JSRR / JMPR este similară cu cea a instrucţiunilor JSR / JMP, singura deosebire constând în faptul că adresa de apel / salt se obţine în cazul primelor prin mod de adresare indexat, iar în cazul celor din urmă prin mod de adresare direct. Această caracteristică importantă permite ca apelul subrutinelor respectiv saltul necondiţionat implicat de instrucţiunile JSRR / JMPR să nu se realizeze neapărat într-o vecinătate a instrucţiunii curente (cazul JSR / JMP – echivalent apelurilor / salturilor de tip NEAR la procesorul INTEL) ci şi în alte zone din spaţiul de memorie utilizator (apeluri / salturi de tip FAR).

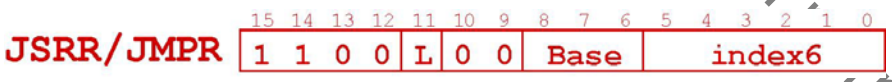


Figura 8.10. Formatul instrucţiunilor JSRR/JMPR

Primii 4 biţi din formatul instrucţiunii JSRR / JMPR sunt 1100, urmaţi de bitul IR[11]=L cu acelaşi rol de selecţie între apel (L=1) şi salt necondiţionat (L=0). Următorii doi biţi 0 iar biţii IR[8:6] codifică registrul de bază folosit în adresare. Deplasamentul este obţinut din câmpul *index6* IR[5:0] extins cu 0 până la 16 biţi. Adresa de apel / salt se obţine prin însumarea conţinutului registrului de bază cu deplasamentul (lucru ilustrat de etapa 2 din figura 8.11). În registrul R7 se încarcă valoarea PC-ului (care

în momentul execuţiei instrucţiunii JSRR indică spre instrucţiunea următoare din programul apelant, PC-ul fiind incrementat în timpul fazei FETCH) – vezi operaţia **1** din figura 8.11. În PC se încarcă adresa primei instrucţiuni din subrutina apelată, predându-se controlul acesteia – vezi operaţia 3 din figura 8.11. Revenirea din subrutină în cazul instrucţiunilor JSR / JSRR se realizează cu ajutorul instrucţiunii RET, care restaurează în PC valoarea aflată în R7 (adresa primei instrucţiuni din subrutina apelantă, succesoare apelului), mecanism similar cu cel întâlnit în cazul instrucţiunilor TRAP. Întrucât în cazul instrucţiunilor JMP / JMPR nu se face apel ci doar salt necondiţionat rezultă că nu se va face uz de instrucţiunea RET, în caz contrar s-ar modifica în mod eronat semantica programului (*pasul 1 din figura 8.11 nu apare în cazul instrucţiunii JMPR*).

Instrucţiunile JSRR / JMPR poartă numele de instrucţiuni de apel / salt cu mod de adresare indirect prin registru. Ele, ca de altfel toate instrucţiunile de salt condiţionat / necondiţionat, apel direct / indirect cauzează hazarduri de ramificaţie în procesarea pipeline a instrucţiunilor. Metodele de soluţionare implementate în procesoarele actuale sunt fie *software* bazate pe reorganizarea codului sursă, dar mai ales *hardware* bazate pe predicţia dinamică a adreselor destinaţie aferente instrucţiunilor de salt. După cum se ştie, predicţia adreselor destinaţie ale salturilor / apelurilor indirecte este o problemă extrem de *dificilă* (întrucât adresele ţintă se modifică în mod dinamic) şi *actuală* iar soluţiile propuse nu sunt încă mulţumitoare. În cadrul unei lucrări anterioare, autorul acestei cărţi a sesizat faptul că, înainte de a aborda efectiv problema predicţiei ar trebui să înţeleagă în profunzime modurile în care aceste apeluri indirecte sunt generate prin compilare. Pentru aceasta, el dezvoltă programe de test proprii, atât procedurale cât şi obiectuale, care conţin corpuri "suspectate" de a genera apeluri şi salturi indirecte în urma compilării. Pe baza acestei cercetări se extrag concluzii valoroase şi deosebit de utile cu privire la construcţiile din programele (limbajele) de nivel înalt (*High Level Languages*) generatoare ale unor astfel de salturi (apeluri indirecte de funcţii prin pointeri, construcţii *switch/case*, prezenţa funcţiilor de bibliotecă, inclusiv biblioteci dinamice de tip DLL, legarea dinamică realizată prin polimorfism etc.). Toate aceste informaţii de nivel semantic superior pot fi utilizate cu succes în predicţia adreselor salturilor / apelurilor indirecte care, actualmente, utilizează în majoritatea lor informaţii de nivel semantic mai scăzut, precum codul obiect, adrese binare etc. Practic, pierderea semanticii codului de nivel înalt după compilare, devine inacceptabilă pentru noua generaţie de microprocesoare.

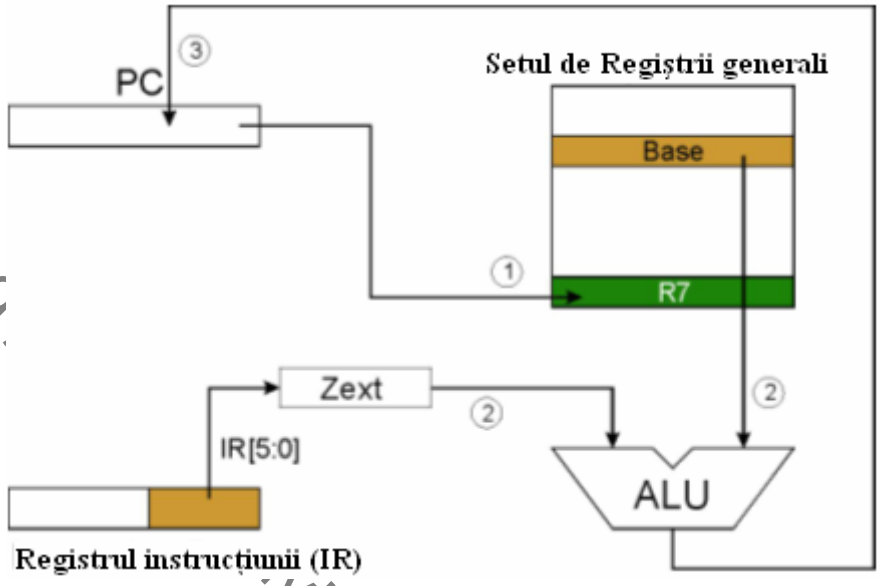


Figura 8.11. Modul de lucru al instrucțiunii **JSRR**

În continuare se prezintă un exemplu de folosire a unei subrutine definite de utilizator. Subrutina de la adresa **2sComp** determină *complementul față de 2* al unui număr transmis ca parametru prin registrul R0, iar rezultatul este returnat tot prin intermediul registrului R0. Programul principal urmăreşte calcularea diferenței dintre regiştrii R1 şi R3. Întrucât scăderea unui operand reprezintă de fapt o adunare cu negatul acestuia (operandul transformat în complement față de 2), în cazul de față, registrul R3 având rolul de scăzător, nu va fi scăzut ci va fi transmis ca argument (parametru efectiv) subrutinei 2sComp. La revenirea din subrutină la descăzut (R1) se adaugă rezultatul returnat de subrutina 2sComp (valoarea – R3). Evident că, pentru o funcționare corectă a programului, dacă valoarea registrului R0 va fi necesară în continuare, atunci programul apelant trebuie să salveze conținutul lui R0 înaintea apelului subrutinei 2sComp.

```
; Calculul complementului față de 2 a valorii stocate în registrul R0
2sComp    NOT   R0, R0      ; inversează fiecare bit al registrului
                            (se determină complementul față de 1
                            a valorii stocate inițial în R0)
          ADD   R0, R0, #1  ; se adună 1 pentru a determina
                              complementul față de 2
          RET               ; revenirea din subrutină
```

```
; Apelul subrutinei din programul principal trebuie făcut din aceeaşi pagină
  cu subrutina (mod de adresare direct)
; Se doreşte calcularea valorii R4 = R1 - R3
        ADD   R0, R3, #0    ; valoarea lui R3 este transmisă ca
                            argument subrutinei 2sComp; se
                            copiază R3 în R0
        JSR   2sComp        ; apelul subrutinei pentru negarea
                            valorii din R0
        ADD   R4, R1, R0    ; scăderea se realizează prin adunarea
                              negatului scăzătorului la descăzut
        ...
```


8.3.2. TRANSFERUL PARAMETRILOR CĂTRE ŞI DE LA SUBRUTINE

Transferul de informaţie de la programul (subrutina) apelant(ă) la subrutina apelată şi invers se realizează prin intermediul **argumentelor** şi respectiv al **valorii returnate**.

Argumentele reprezintă una sau mai multe valori, transmise subrutinei apelate prin intermediul regiştrilor, necesare acesteia pentru a-şi îndeplini task-ul. Exemple de argumente la LC-3 ISA:

- În subrutina 2sComp, R0 reprezintă numărul ce trebuie negat (valoarea care trebuie transformată în complement faţă de 2).
- În rutina de serviciu aferentă apelului sistem OUT, R0 reprezintă codul ASCII al caracterului ce va fi afişat.
- În rutina de serviciu aferentă apelului sistem PUTS, R0 reprezintă adresa şirului de caractere ce va fi afişat.

Valoarea returnată constituie rezultatul calculat de subrutină (un număr sau o adresă a unui tablou de numere sau de caractere). Exemple:

- În subrutina 2sComp, valoarea negată (complementul faţă de 2 a valorii primite ca parametru) este transmisă prin registrul R0.
- În rutina de serviciu GETC, codul ASCII al caracterului citit de la tastatură este returnat prin R0.

Pentru a putea folosi o subrutină, programatorul trebuie să cunoască:

- Adresa acesteia (valoare numerică sau simbolică – etichetă, care identifică prima instrucţiune din subrutină).
- Ce realizează subrutina (scopul, funcţia, rezultatul acesteia).

Aşa cum s-a amintit anterior referitor la avantajul utilizării subrutinelor şi distribuirea sarcinilor între programatori, un programator nu

trebuie să cunoască neapărat cum funcționează subrutina (în interiorul său), ci doar ce modificări sunt vizibile la nivelul stării arhitecturale a maşinii (procesorului), după execuția subrutinei. De asemenea, trebuie ştiut argumentele şi respectiv, dacă există, valoarea returnată.

8.3.3. SALVAREA ŞI RESTAUREA REGIŞTRILOR ÎN CAZUL SUBRUTINELOR. RUTINE (FUNCȚII) DE BIBLIOTECĂ

La fel ca şi în cazul utilizării rutinelor de serviciu aferente întreruperilor software, regiştrii trebuie salvați şi restaurați la valorile anterioare apelului. *În aproape toate cazurile de utilizare a subrutinelor este folosită strategia „callee-save", excepție făcând salvarea regiştrilor care reprezintă valoarea returnată*. Sunt salvați toți regiştrii care vor fi alterați pe parcursul execuției subrutinei, acest lucru nefiind vizibil la nivelul apelantului. Restaurarea argumentelor de intrare în subrutină la valoarea anterioară apelului este o practică uzuală. Dacă argumentele sunt transmise însă subrutinei prin referință (adresă) atunci valorile acestora pot fi modificate în interiorul acesteia şi să rămână aşa la revenirea din subrutină (pentru mai multe detalii vezi capitolul 12). **Atenție** însă, înaintea apelului oricărei subrutine folosind instrucțiunile JSR sau JSRR, sau înaintea oricărui apel sistem (instrucțiune TRAP) valoarea registrului R7 trebuie salvată, în caz contrar revenirea nu se va mai face la nivelul apelantului.

Pe lângă subrutinele definite de utilizator în programele de calcul se întâlnesc **biblioteci de rutine cu funcții specifice**: formule matematice, dedicate aplicațiilor grafice, aferente dispozitivelor periferice, anumitor componente ale sistemului de calcul sau sistemului de operare. Rutinele de bibliotecă sunt definite în afara programului utilizator (independent de acesta) şi sunt apelate în programele utilizator. Pentru aceasta, în LC-3 ISA, declararea în programul utilizator a adresei de start (de regulă simbolice) a unei rutine de bibliotecă se face folosind directiva *.EXTERNAL*. Fiecare rutină de bibliotecă este caracterizată de propria tabelă de simboluri. Link-editorul este utilitarul (aplicația) responsabil(ă) cu rezolvarea (identificarea corectă) a fiecărei adrese declarată .EXTERNAL înaintea creării imaginii executabile a aplicației. Comercianții de aplicații software (sisteme de operare, compilatoare, aplicații dedicate, drivere pentru diferite componente ale sistemului de calcul, etc.) pun la dispoziția utilizatorilor fişiere obiect (precompilate) conținând diverse rutine de bibliotecă. Din motive de proprietate intelectuală nu se asigură codul sursă al acestora. De asemenea, trebuie ca în programul apelant să se realizeze apelul (cel mai probabil) prin

instrucțiuni JSRR, iar asamblorul şi link-editorul să suporte declarații de etichete .EXTERNAL.

În continuare este prezentat un exemplu care utilizează rutina de bibliotecă matematică de extragere a radicalului.

```
                ...
                .EXTERNAL SQRT      ; rutina matematică de
                                    extragere a radicalului este
                                    definită în alt modul, probabil
                                    la un deplasament mai mare de
                                    256 adrese (crescător sau
                                    descrescător relativ la PC-ul
                                    curent)
                ...
                LD   R2, SQAddr     ; încarcă adresa de început a
                                    subrutinei SQRT
                JSRR R2, #0         ; realizează apelul indirect
                ...
SQAddr          .FILL   SQRT
```

8.4. EXERCIȚII ŞI PROBLEME

1. Se consideră următorul program scris în limbaj de asamblare LC-3:

```
        .ORIG x3000
L1      LEA    R1, L1
        AND R2, R2, x0
        ADD R2, R2, x2
        LD  R3, P1

        JSR L2
        BRnzp NEXT
L2      LDR R0, R1, x16
        ST  R7, SAVER7
        OUT
        LD  R7, SAVER7
        ADD R3,R3,-1
```

```
        BRz GLUE
        ADD R1, R1, R2
        BRnzp L2
GLUE    RET
NEXT    LEA    R1, L1
        ADD    R1, R1, #1
        LD R3, P1
        ADD R3, R3, #-7
        JSR L2
        HALT
P1      .FILL x12
        .STRINGZ "SLaArMbUaLtToIrAiNFIe!rriCcaizt!ea!"
SAVER7  .BLKW 1
        .END
```

a) Generați tabela de simboluri aferentă acestui program.
b) Scrieți codificarea binară a instrucțiunilor aflate la următoarele adrese: x3000, x3004, x3008, x300B, x300E, x3012.
c) Ce afişează programul ? De ce sunt necesare cele două instrucțiuni care îl salvează şi restaurează temporar pe R7 ?

2. Răspundeți la întrebările:
a) Câte rutine de serviciu pot fi implementate prin intermediul instrucțiunilor de apel sistem TRAP în cadrul arhitecturii LC-3 ? De ce ?
b) De ce este necesară utilizarea unei instrucțiuni RET pentru revenirea dintr-o rutină TRAP (de tratare a unei întreruperi)? De ce nu poate fi folosită în schimb o instrucțiune de salt necondiționat BRnzp ?
c) Câte accese la memorie se fac în timpul procesării unei instrucțiuni TRAP, presupunând că ea a fost adusă deja în registrul IR (s-a efectuat faza FETCH) ?
d) Poate o rutină de tratare (nu doar cele implementate la LC-3 ISA ci în cazul general) să apeleze o altă rutină de tratare ? În caz afirmativ, există vreo acțiune specială care trebuie realizată la nivelul rutinei apelante ? Justificați.
e) Există vreun apel sistem care să repornească ceasul sistemului ?

3. Se consideră următorul program scris în limbaj de asamblare LC-3:

```
        .ORIG x3000
        LEA   R0, DATA
        AND   R1, R1, #0
        ADD   R1, R1, #9
LOOP1
        ADD   R2, R0, #0
        ADD   R3, R1, #0
LOOP2
        JSR   SUB
        ADD   R2, R2, #1
        ADD   R3, R3, #-1
        BRP   LOOP2
        ADD   R1, R1, #-1
        BRP   LOOP1
        HALT
DATA  .BLKW #10
SUB   LDR   R5, R2, #0
        NOT   R4, R5
        ADD   R4, R4, #1
        LDR   R6, R2, #1
        ADD   R4, R4, R6
        BRZP   CONT
        STR   R5, R2, #1
        STR   R6, R2, #0
CONT RET
     .END
```

Presupunem că locaţiile de memorie începând cu adresa de la eticheta DATA conţin 10 numere întregi în complement faţă de 2. Aceste numere trebuie setate de utilizatorul programului înainte de execuţie. Ce realizează programul ? Care este relaţia dintre valorile iniţiale şi cele finale de la adresele respective ?

4. Intenţia următorului program este de a afişa valoarea 5 pe consolă (ecran). Din nefericire nu se întâmplă acest lucru. De ce ? Răspundeţi în cel mult 10 cuvinte.

```
        .ORIG x3000
        JSR     A
        OUT
        BRnzp DONE
```

```
A       AND R0, R0, #0
        ADD R0, R0, #5
        JSR  B
        RET
DONE    HALT
ASCII   .FILL 0x0030
B       LD R1, ASCII
        ADD R0, R0, R1
        RET
        .END
```

5. Răspundeţi la întrebările:
   a) Care sunt diferenţele dintre instrucţiunile TRAP şi JSRR ? Dar dintre JSR şi JSRR ?
   b) Există vreo instrucţiune care face acelaşi lucru ca şi instrucţiunea JMP ?
   c) Care este scopul apelurilor sistem ?
   d) Care este diferenţa dintre strategiile "*caller-save*" şi "*callee-save*" în salvarea şi restaurarea regiştrilor la apelul şi revenirea din subrutine.
   e) Ce anume (dacă există ceva) este greşit în următoarea subrutină ?

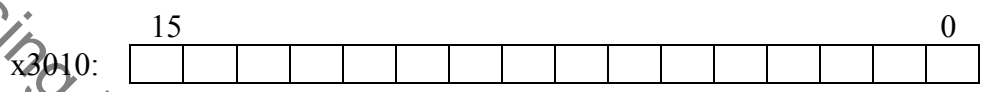
```
DO      JSR  DOTHIS
        JSR  DOTHAT
        RET
```

6. În tabelul de mai jos sunt ilustrate conţinuturile regiştrilor arhitecturii LC-3 înainte şi după execuţia instrucţiunii de la adresa 0x3010. Identificaţi instrucţiunea memorată la adresa 0x3010. De notat că, există suficientă informaţie în tabelul specificat care să personalizeze instrucţiunea.

	Înainte	**După**
R0	xFF1D	xFF1D
R1	x321C	x321C
R2	x2F11	x2F11
R3	x5321	x5321
R4	x331F	x331F
R5	x1F22	x1F22
R6	x01FF	x01FF
R7	x341F	x3011
PC	x3010	x3220

N	0	0
Z	1	1
P	0	0

Completați instrucțiunea lipsă în căsuțele libere:



7. Cele două secvențe de mai jos sunt asamblate separat. În urma acestui proces sunt detectate două erori: una în faza de asamblare şi una în faza de link-editare. Încercuiți fiecare greşeală şi stabiliți în ce fază a fost detectată.

```
        .EXTERNAL
SQRT
        .ORIG   0x3000
        LD R0, VALUE
        JSR     SQRT
        ST R0, DEST
        HALT
VALUE       .FILL 0x30000
DEST        .FILL 0x0025
        .END
```

```
        .ORIG 0x3200
SQRT    ADD R0, R0, #0
        ;codul necesar
        implementării funcției de
        extragere a rădăcinii
        pătrate
        ;returnează rezultatul în R0
        RET
        .END
```

8.

a) Completați tabela de simboluri pentru următorul program:

```
            .ORIG   x3000
            LEA     R6, STACKBASE
            ADD     R6, R6, #-1
            LEA     R0, PROMPT
            TRAP    x22             ; PUTS
            AND     R1, R1, #0
LOOP        TRAP    x20             ; IN
            TRAP    x21
```

```
                ADD     R3, R0, x-A     ; negația (valoarea în complement față de 2) a codului ASCII pentru tasta "Enter"
                BRz     INPUTDONE
                JSR     PUSH
                ADD     R1, R1, #1
                BRnzp   LOOP
INPUTDONE       ADD     R1, R1, #0
                BRz     DONE
LOOP2           JSR     POP
                TRAP    x21
                ADD     R1, R1, #-1
                BRp     LOOP2
DONE            TRAP    x25             ; HALT

PUSH            ADD     R6, R6, #1
                STR     R0, R6, #0
                RET

POP             LDR     R0, R6, #0
                ADD     R6, R6, #1
                RET

PROMPT          .STRINGZ "Please enter a sentence: "
STACKBASE       .BLKW   #50
                .END
```

Tabela de Simboluri

Eticheta (*label*)	Adresa

b) Ce realizează programul anterior ? Explicați rolul fiecărui registru folosit: R0, R1, R3 şi R6. Mai există şi alți regiştrii care îşi modifică valoarea pe parcursul execuției programului ?

9. În secvența de program de mai jos lipsesc câteva instrucțiuni. Semantica programului este următoarea: utilizatorului îi este cerut să îşi introducă numele. Programul memorează mesajul „Hello, ” urmat de numele introdus, ca un şir de caractere începând de la adresa indicată de simbolul HELLO şi-l afişează pe consolă. Se presupune că utilizatorul a încheiat introducerea numelui său de la tastatură prin Enter, al cărui cod ASCII este 0x0A. Numele este restricționat la maximum 25 de caractere. De exemplu, presupunând că utilizatorul a introdus de la tastatură numele John, atunci pe consolă va apărea mesajul:

Please enter your name: John
Hello, John

Se cere să se introducă în secvență instrucțiunile lipsă.

```
            .ORIG   x3000
            LEA     R1, HELLO
AGAIN       LDR     R2, R1, #0
            BRz     NEXT
            ADD     R1, R1, #1
            BRnzp   AGAIN
NEXT        LEA     R0, PROMPT
            TRAP    0x22                ; PUTS

            ____________________

AGAIN2      TRAP    0x20                ; GETC
            TRAP    0x21                ; OUT
            ADD     R2, R0, R3
            BRz     CONT

            ____________________

            ____________________

            BRnzp   AGAIN2
CONT        AND     R2, R2, #0

            ____________________

            LEA     R0, HELLO
            TRAP    0x22                ; PUTS
```

```
                TRAP    0x25                ; HALT
NEGENTER        .FILL   0xFFF6              ; -0x0A
PROMPT          .STRINGZ „Please enter your name: ”
HELLO           .STRINGZ „Hello, ”
                .BLKW    #25
                .END
```

10. Programului de mai jos îi lipsesc câteva instrucțiuni importante! Dacă sunt completate corect atunci programul va afişa pe ecran mesajul: ABCFGH. Completați instrucțiunile lipsă astfel încât programul original să realizeze ceea ce şi-a propus. În fiecare spațiu lipseşte exact o instrucțiune iar toate celelalte instrucțiuni sunt corecte.

```
                .ORIG x3000
                LEA R1, TESTOUT
BACK_1          LDR R0, R1, #0
                BRz NEXT_1
                TRAP x21

                _____________
                BRnzp BACK_1
;
NEXT_1          LEA R1, TESTOUT
BACK_2          LDR R0, R1, #0
                BRz NEXT_2
                JSR SUB_1
                ADD R1, R1, #1
                BRnzp BACK_2
;
NEXT_2          _____________
;
SUB_1           _____________
K               LDI R2, CRTSR

                _____________
                STI R0, CRTDR
                RET
CRTSR           .FILL xF3FC
CRTDR           .FILL xF3FF
TESTOUT         .STRINGZ "ABC"
                .END
```

9. STIVA – STRUCTURĂ. PRINCIPIU DE FUNCȚIONARE. OPERAȚII AFERENTE (*PUSH & POP*)

9.1. STRUCTURA DE DATE DE TIP STIVĂ

9.1.1. PRINCIPIU DE FUNCȚIONARE

Conceptul de **stivă** vine să încheie partea de introducere în ştiința şi ingineria calculatoarelor strict legată de hardware (de ISA aferentă unui procesor). Din punct de vedere fizic, stiva reprezintă o importantă structură de memorare folosită pentru implementarea la nivel hardware a concepte software (mecanisme) fundamentale: *apelul funcțiilor*, *recursivitate*. Definitoriu pentru stivă reprezintă modul de acces la elementele sale şi nu modul specific de implementare. Din punct de vedere software stiva este o structură de date abstractă definită prin regulile de inserare şi extragere a datelor în / din ea, ambele operații fiind efectuate la acelaşi capăt. Principiul de memorare este de tip **LIFO (last-in first-out)**, adică ultimul inserat va fi primul extras din stivă, elementele extrase apărând practic în ordine inversă față de cum au fost introduse.. Analizând stiva pe o perioadă îndelungată se poate spune, chiar dacă nu este tot timpul adevărat, şi că primul introdus în stivă este ultimul care va fi scos.

9.1.2. IMPLEMENTARE HARDWARE ŞI SOFTWARE.

Din punct de vedere fizic, cel mai intuitiv mod de a înțelege stiva îl reprezintă o stivă de farfurii sau mai bine, un resort care reține monede una peste cealaltă. Şi într-un caz şi în celălalt **accesul se face doar la elementul** (farfuria / moneda) **din vârf**. Dacă se dorește accesul la oricare alt element trebuie extrase din stivă elementele din vârf până la cel solicitat.

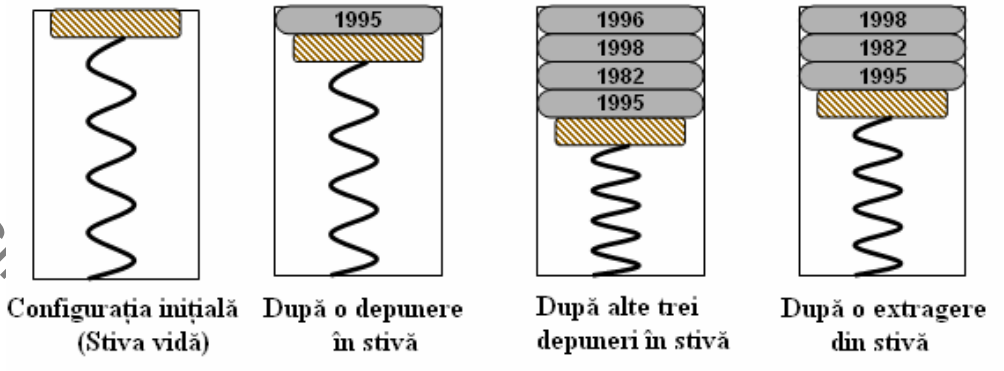


Figura 9.1. Exemplu de stivă. Model fizic

În figura 9.1 sunt prezentate operațiile cu stiva din perspectiva unui model fizic (un resort cu monede). Ordinea operațiilor este de la stânga la dreapta. În faza inițială stiva fiind vidă resortul este relaxat (necomprimat). Prima depunere în stivă este cea a monedei de etichetă 1995. Urmează alte trei depuneri: întâi moneda etichetată 1982, apoi cea 1998 şi ultima 1996. Imaginea cea mai din dreapta reprezintă configurația stivei după extragerea monedei din vârful stivei (a monedei etichetate 1996). Noul vârf al stivei are eticheta 1998.

În continuare (vezi figura 9.2) este descrisă implementarea în hardware a stivei de monede, folosind regiştrii. În acest caz, stiva este alcătuită dintr-un număr de regiştrii hardware aflați în poziții bine determinate (fixe). La depunerea a câte unui element în stivă dacă stiva nu este goală atunci informația este translatată (în jos) spre baza stivei (copiată dintr-un registru în altul), iar în vârful stivei se copiază noua valoare inserată. Se observă că **implementarea stivei de monede în hardware** folosind regiştrii **implică deplasarea informației între regiştrii inacceptabil din punct de vedere al memoriei implementată fizic** (este ineficient din punct de vedere al procesării informației ca fiecare scriere în memorie – depunere în stivă – să implice transferul tuturor informațiilor existente la acel moment între locații succesive de memorie).

În figura 9.2 sunt prezentate operațiile cu stiva din perspectiva unui model hardware (stiva compusă din regiştrii). Ordinea operațiilor este de la stânga la dreapta. Inițial stiva este vidă (Funcția Empty ilustrează dacă există elemente în stivă. În acest caz ea va returna Da). Prima operație o reprezintă inserarea în stivă a valorii 18 (Funcția Empty va genera Nu). Urmează trei depuneri succesive în stivă (făcând referire la figura 9.1 se poate spune că resortul se comprimă). Se depun în stivă pe rând valorile: 31, 5 şi apoi 12 (evident că funcția Empty returnează Nu). Colțul din dreapta al figurii ilustrează conținutul stivei după două operații succesive de extragere

din stivă (a elementelor din vârful stivei – întâi 12 şi apoi 5). Trebuie observat că vârful stivei (TOP – *top of the stack*) – indicator la ultima locaţie ocupată din stivă, rămâne tot timpul în acelaşi loc. Întrucât în stivă mai rămân valorile 31 şi 5 funcţia Empty va genera Nu.

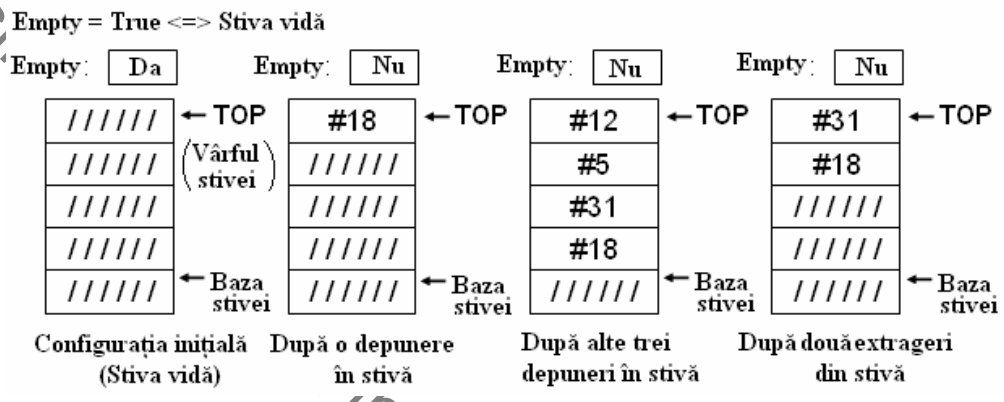


Figura 9.2. Implementarea stivei de monede în hardware folosind regiştrii

Implementarea software a stivei are la bază ideea de a nu permite deplasarea informaţiei în memorie la fiecare acces la stivă. Doar vârful stivei poate să-şi modifice poziţia, el fiind reţinut într-un singur registru.

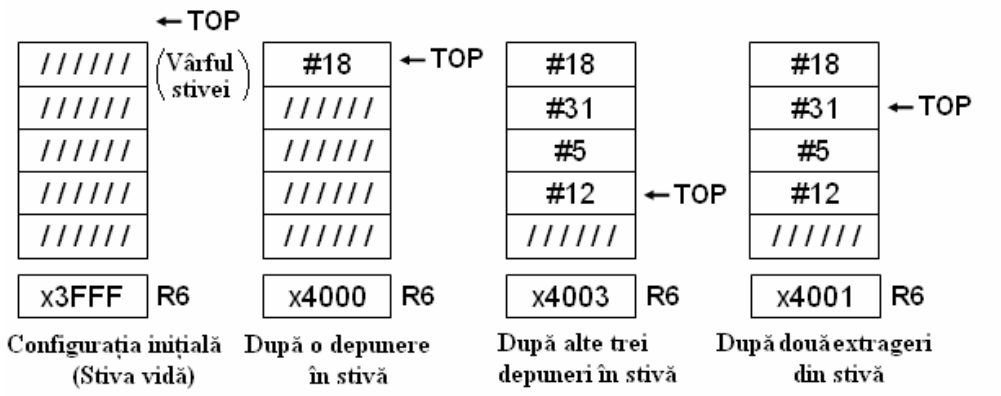


Figura 9.3. Implementarea software a stivei

Prin convenţie la LC-3 ISA [Patt03] registrul R6 reţine adresa vârfului stivei (reprezintă un indicator spre ultima locaţie ocupată din stivă). La alte arhitecturi (Intel, MIPS) acest registru se numeşte SP (*stack pointer* – vezi capitolul 12 pentru clarificarea noţiunii de pointer).

9.1.3. OPERAȚII AFERENTE (PUSH & POP)

Asupra unei stive se definesc câteva operații, dintre care cele mai importante sunt:

1. Depune un element pe stivă (PUSH);
2. Extrage un element din stivă (POP);

Cele două operații se realizează în vârful stivei. Astfel, dacă se scoate un element din stivă, atunci acesta este cel din vârful stivei şi în continuare, cel pus anterior lui pe stiva ajunge în vârful stivei. Dacă un element se pune pe stivă, atunci acesta se pune în vârful stivei şi în continuare el ajunge în vârful stivei. La eliminarea tuturor elementelor din stivă aceasta devine *vidă*.

Din punct de vedere software implementarea unei stive poate fi făcută printr-o listă simplu înlănțuită. Trebuie însă identificate baza şi vârful stivei cu ajutorul capetele listei simplu înlănțuite. Exista două posibilități:

a. Nodul spre care indică (pointează) variabila *prim* este baza stivei, iar nodul spre care indică variabila *ultim* este vârful stivei;
b. Nodul spre care pointează variabila *prim* este vârful stivei, iar nodul spre care pointează variabila ultim este baza stivei.

În continuare este descrisă implementarea în LC-3 ISA a codului aferent operațiilor de bază de depunere în stivă (Push) şi de extragere din stivă (Pop). De menționat că, procesoarele Intel au definite instrucțiuni cu acelaşi nume *PUSH reg şi POP reg*. Procesoarele RISC însă (denumite şi maşini load/store) implementează operațiile cu stiva prin instrucțiuni de scriere şi citire din memorie (*store* – pentru Push) şi (*load* – pentru Pop).

Push

```
ADD R6, R6, #1      ; incrementează valoarea indicatorului de stivă
                      (se modifică TOS).
STR R0, R6, #0      ; memorează (depune în stivă) data aflată în
                      registrul R0.
```

Pop

```
LDR R0, R6, #0      ; citeşte din memorie data de la adresa
                      specificată de TOS
ADD R6, R6, #-1     ; decrementează indicatorul de stivă (se
                      modifică TOS).
```

Obs: De remarcat că, deşi în cazul unei stive fizice (de farfurii – de exemplu), la extragerea din stivă elementul dispare, în urma operației de *POP* din stivă data se preia din memorie într-un registru destinație, dar informația rămâne la adresa respectivă. Actualizarea vârfului stivei

este operaţia ce marchează extragerea unui element din stivă permiţându-se în continuare încărcarea altuia la locaţia „*recent eliberată*".

Condiţiile de eroare ce pot să apară sunt:

- *Underflow* (Încercarea de a extrage un element dintr-o stivă vidă).
- *Overflow* (Încercarea de a depune pe stivă plină un element).

Pentru evitarea apariţiei acestor condiţii de eroare trebuie monitorizată adresa vârfului stivei (top of stack – *TOS*). În cadrul LC-3 ISA, vârful stivei reţine tot timpul adresa ultimei locaţii ocupate din stivă (fiecare operaţie Push incrementează TOS iar fiecare operaţie Pop îl decrementează). Astfel înaintea fiecărei operaţii de depunere în stivă trebuie întâi actualizat vârful stivei şi apoi scrisă informaţia la noua adresă (noul TOS) iar la fiecare extragere din stivă trebuie, după citirea datei din stivă, să se actualizeze TOS. Stiva creşte în capacitate (în locaţii utilizate) în jos. La anumite procesoare (MIPS pe platformă Linux) stiva creşte în capacitate de la adrese mari spre adrese mici (deci registrul SP va fi decrementat la fiecare operaţie Push şi incrementat la fiecare Pop).

Practic înaintea operaţiei de depunere în stivă trebuie testată condiţia de *overflow* iar la extragere trebuie testată condiţia de *underflow*. Schema logică de mai jos verifică dacă înaintea unei extrageri stiva este vidă şi marchează în registrul R5 rezultatul (1 dacă apare *underflow* şi 0 dacă se poate face extragerea, iar în R0 se depune valoarea din vârful stivei).

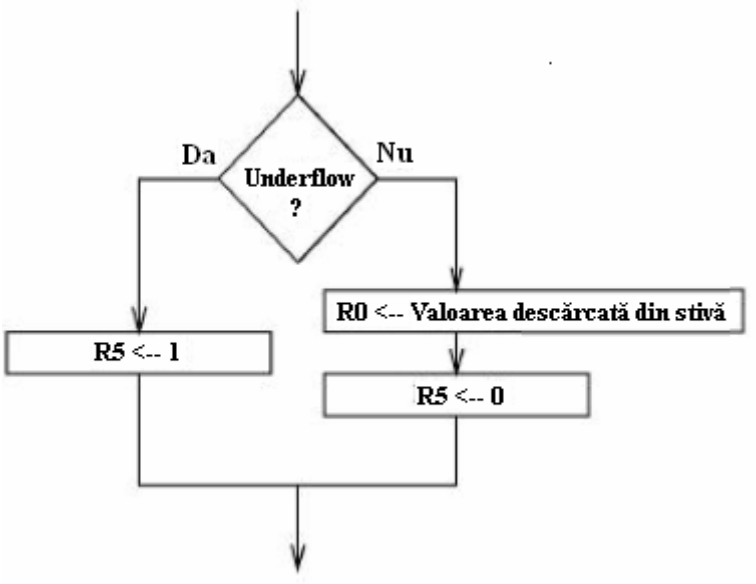


Figura 9.4 Testarea condiţiei de *underflow*

Astfel prin introducerea instrucțiunilor care testează condițiile de eroare codul echivalent instrucțiunilor PUSH şi POP se complică. În continuare sunt descrise două subrutine care implementează cele două operații cu stiva.

Codul aferent operației POP cu verificarea condiției de *Underflow*

; se consideră implementarea software a stivei din figura 9.3 în care valoarea indicatorului de stivă (R6) pentru stiva vidă este 0x3FFF.

```
POP     LD  R1, EMPTY       ; EMPTY = -0x3FFF (inversul
                              indicatorului de stivă în cazul în care
                              stiva vidă)
        ADD R2, R6, R1        ; Se verifică dacă stiva este vidă
        BRz FAIL              ; În caz afirmativ se trece la setarea
                              condiției pentru eroare
        LDR R0, R6, #0        ; Stiva nefiind vidă se încarcă în R0
                              valoarea din vârful acesteia
        ADD R6, R6, #-1       ; Se decrementează indicatorul de stivă
        AND R5, R5, #0        ; Se indică SUCCESS prin resetarea
                              registrului R5
        RET
FAIL    AND R5, R5, #0              ; FAIL: R5 = 1
        ADD R5, R5, #1        ; Se încarcă în R5 valoarea 1 pentru a
                              marca eroare la descărcare din stivă
        RET
EMPTY   .FILL xC001
```

Codul aferent operației PUSH cu verificarea condiției de *Overflow*

; Se consideră implementarea software a stivei din figura 9.3 cu 5 valori care pot fi depuse în stivă. Indicatorul de stivă (R6) pentru stiva vidă este 0x3FFF. Secvența de mai jos verifică dacă înaintea unei depuneri pe stivă aceasta este plină (nu mai pot fi depuse alte elemente) şi marchează în registrul R5 rezultatul (1 dacă apare *overflow* şi 0 dacă se poate face depunerea, valoarea care va fi scrisă în vârful stivei fiind cea aflată în registrul R0).

```
PUSH  LD  R1, MAX         ; MAX = -x4004
        ADD R2, R6, R1          ; Compară stack pointer-ul cu x3FFF
        BRz FAIL
```

```
        ADD R6, R6, #1      ; Vârful stivei va fi incrementat, stiva
                            crescând în capacitate
        STR R0, R6, #0
        AND R5, R5, #0 ; SUCCESS: R5 = 0
        RET
FAIL    AND R5, R5, #0 ; FAIL: R5 = 1
        ADD R5, R5, #1
        RET
MAX         .FILL 0xBFFC
```


9.1.4. MODUL DE LUCRU PRIN ÎNTRERUPERI HARDWARE.

Una din facilităţile mecanismului de stivă o reprezintă posibilitatea apelării funcţiilor imbricate (recursivitate directă sau indirectă) şi după cum se va vedea în continuare al tratării întreruperilor imbricate (apariţia unei întreruperi în chiar rutina de serviciu (tratare) a altei întreruperi, atunci când este permis). După cum s-a mai specificat şi în capitolul 8, o întrerupere hardware poate fi considerată un apel de subrutină (spre necunoscut din punct de vedere al programatorului dar nu şi al procesorului şi al sistemului de operare) declanşat asincron de un eveniment extern.

Modul de lucru prin întreruperi hardware urmăreşte tratarea eficientă din punct de vedere al performanţei de procesare a interacţiunilor dintre dispozitivele periferice şi procesor. De câte ori doreşte un serviciu de la procesor, dispozitivul extern, prin interfaţa sa, lansează un semnal de întrerupere (de exemplu, când se introduce un caracter de la tastatură sau când monitorul / imprimanta sunt gata pentru a afişa un nou caracter). În urma recepţionării semnalului de întrerupere, procesorul încheie ciclul instrucţiunii aflate în execuţie şi părăseşte programul principal urmând să intre în rutina de serviciu a întreruperii respective (similară rutinelor de serviciu aferente întreruperilor software – instrucţiunilor TRAP la LC-3 ISA) pentru satisfacerea cererii respectivului dispozitiv extern. În cele mai multe cazuri, la încheierea rutinei de tratare a întreruperii se predă controlul programului utilizator (cu ajutorul instrucţiunii RTI la LC-3 ISA, IRET la Intel sau JR $31 la MIPS), reluarea procesării făcându-se cu instrucţiunea imediat următoare celei pe care a apărut întreruperea. Pot apărea însă cazuri în care, în timpul tratării unei întreruperi apare o nouă cerere de întrerupere din partea altui dispozitiv periferic. Dacă este permisă întreruperea (vezi într-un exemplu ulterior) atunci se intră în rutina de tratare a noii întreruperi.

La încheierea acesteia se revine în rutina de tratare a primei întreruperi care se va procesa până la terminare, în final cedându-se controlul programului principal.

Avantajul modului de lucru prin întreruperi hardware constă în faptul că procesorul nu mai trebuie să interogheze la anumite intervale de timp fiecare dispozitiv periferic, putând executa alte sarcini în acest timp (instrucțiuni utile).

Dintre cauzele apariției întreruperilor hardware se disting următoarele:

- Evenimentele externe sunt nesigure şi au o apariție neregulată (de exemplu, sosirea pachetelor de date din rețea, apăsarea tastelor).
- Operațiile cu dispozitivele externe durează foarte mult (de exemplu, transferul datelor pe/de pe disc în/din memorie) iar procesorul poate face altceva pe perioada transferului (cazul transferurilor cu acces direct la memorie).
- Evenimente rare dar critice: incendii sau alte calamități, situații în care este necesară oprirea sistemelor de calcul.

Figura 9.5 prezintă interacțiunea dintre procesor şi un dispozitiv extern. Când respectivul periferic vrea să întrerupă activitatea procesorului şi să îi ceară acestuia un serviciu, prin intermediul interfeței sale (*controller*) va seta semnalul de întrerupere externă (**INT**=1). Toate acțiunilor executate de către procesor din momentul apariției semnalului de întrerupere INT până în momentul procesării primei instrucțiuni din rutina de tratare a întreruperii formează aşa numitul **protocol hardware de acceptare a întreruperii** desfăşurat în următoarele etape succesive:

1. În general, procesoarele verifică activarea întreruperilor la finele ultimului ciclu aferent instrucțiunii în curs de execuție [Mus97, Lun05] (în cazul LC-3 ISA [Patt03] între faza *STORE RESULT* a instrucțiunii aflate în execuție şi faza *FETCH* instrucțiune a celei care urmează). Odată sesizată întreruperea INT de către procesor acesta îşi va termina instrucțiunea in curs de execuție după care, dacă nu există activată o cerere de întrerupere sau de magistrală mai prioritare, nu va face faza *FETCH* a următoarei instrucțiuni ci va trece la pasul 2.
2. Identificarea întreruperii: procesorul va inițializa aşa numitul **ciclu de achitare a întreruperii**. Pe parcursul acestui ciclu extern va genera un semnal de răspuns (achitare) a întreruperii **INTA** (*interrupt acknowledge*) spre toate interfețele de intrare - ieşire. Ca urmare a recepționării INTA interfața care a întrerupt va furniza procesorului prin intermediul bus-ului de date o succesiune de 8 biți reprezentând **vectorul de întrerupere** (**vector**). Acest vector diferă pentru fiecare periferic în parte, identificându-l într-un mod unic. Similar cu cazul instrucțiunilor TRAP, vectorul de întrerupere reprezintă un index în

tabela cu adresele de start a rutinelor de serviciu. Informaţia de la locaţia specificată reprezintă adresa primei instrucţiuni din rutina de tratare a întreruperii, valoare care se încarcă în registrul PC. Vectorilor de întrerupere diferiţi îi vor corespunde adrese de rutine diferite.

3. Procesorul va salva pe stivă PC-ul aferent instrucţiunii imediat următoare instrucţiunii executate de către procesor din programul principal (adresa de revenire), pentru a putea şti la finele rutinei de tratare a întreruperii unde să revină exact în programul principal. De asemenea, sunt salvate pe stivă codurile de condiţie (N, Z, P) – vezi exemplul din figurile 9.7÷9.10. Acest fapt se poate dovedi insuficient având în vedere că in cadrul rutinei de serviciu pot fi alteraţi anumiţi regiştri interni ai procesorului – alterare care poate fi chiar dezastruoasă la revenirea în programul principal. Din acest motiv cade in sarcina celui care scrie rutina de serviciu să salveze (folosind accese de depunere în stiva – PUSH) respectiv să returneze corespunzător (folosind accese de extragere din stiva – POP) aceşti regiştrii.
4. După încheierea rutinei de tratare a întreruperii, instrucţiunea RTI restaurează codurile de condiţie şi vechiul PC (adresa de revenire). Este necesara practic o altă instrucţiune de revenire deoarece instrucţiunea RET preia doar adresa de revenire din registrul R7 şi o depune in PC, nu si codurile de condiţie.

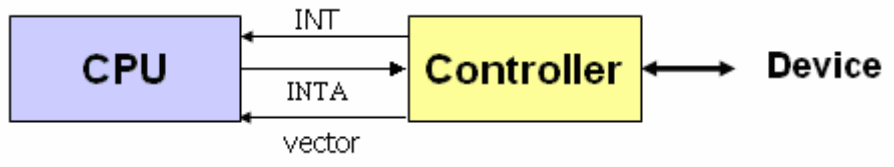


Figura 9.5. Modul de lucru prin întreruperi hardware.

În cazul în care mai mult de un dispozitiv extern doreşte să întrerupă activitatea procesorului în acelaşi timp, procesorul trebuie să asigure un mecanism de prioritizare a cererilor de întrerupere (deservirea perifericelor respective).

În continuare, se exemplifică folosind registrul de stare al tastaturii (KBSR – *keyboard status register*) cum se generează semnalul INT de întrerupere a activităţii procesorului prin apăsarea unei taste de către utilizator. La majoritatea dispozitivelor de intrare / ieşire registrul de stare conţine pe lângă bitul *Ready* (care marchează disponibilitatea perifericului în generarea unei întreruperi) şi un bit de validare a întreruperii de către procesor (*Interrupt Enable*) – vezi figura 9.6. Practic la apăsarea unei taste bitul Ready al KBSR este setat (devine 1). La fel, în cazul în care circuitele

electronice asociate monitorului au încheiat cu succes afișarea ultimului caracter, bitul Ready al CRTSR (registrul de stare al monitorului) devine 1. Cu toate că întreruperea este solicitată de către dispozitivul periferic, în realitate, procesorul nu poate fi întrerupt, fără acceptul acestuia. Bitul 14 al KBSR (IE) poate fi setat sau șters de către procesor, în funcție de disponibilitatea acestuia de a fi întrerupt de către respectivul periferic. Unele procesoare dețin un registru de validare a întreruperilor numit „mască de întreruperi" (*Interrupt Mask*). LC-3 ISA nu deține un astfel de registru. Dacă procesorul acceptă să fie întrerupt atunci, în momentul în care bitul Ready este setat se generează semnalul INT (cerere de întrerupere a activității procesorului).

Se observă că: **INT = IE AND Ready**. Practic, nu e suficient ca utilizatorul să apese o tastă dacă procesorul nu acceptă să fie întrerupt (fiind de exemplu, într-o rutină de serviciu neinteruptibilă, mai prioritară), dar de asemenea, nu e suficient ca procesorul să valideze întreruperile ci trebuie ca și „*ceva să se întâmple*" (se apasă o tastă, se introduce un CD în unitatea CD-ROM spre citire, trebuie imprimat un text pe ecran sau imprimantă, etc.).

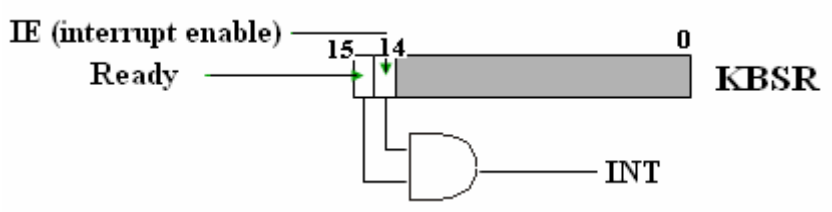


Figura 9.6. Generarea cererii perifericului de întrerupere a activității procesorului

În continuare sunt prezentate o secvență de ipostaze care justifică **necesitatea stivei în cazul tratării întreruperilor hardware imbricate**. In figura 9.7 programul principal A execută instrucțiunea de adunare ADD de la adresa 0x3006, moment în care este întrerupt de către perifericul B.

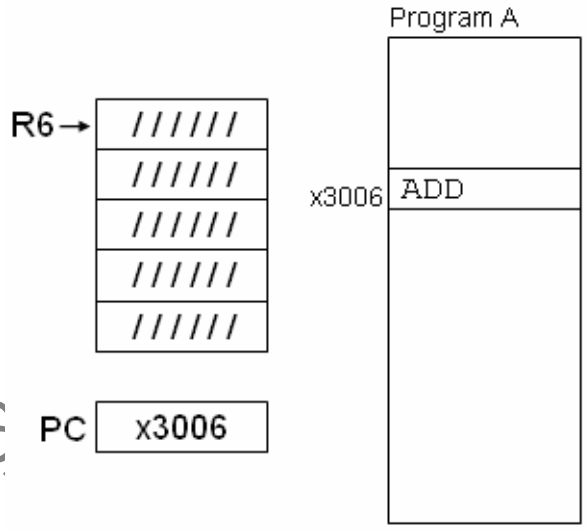


Figura 9.7. Programul principal A execută instrucţiunea ADD de la adresa 0x3006 când perifericul B îl întrerupe

La apariţia semnalului INT, procesorul încheie instrucţiunea de adunare ADD, salvează pe stivă adresa următoarei instrucţiuni din programul principal (PC=0x3007) şi codurile de condiţie tocmai setate de rezultatul adunării şi încarcă în PC valoarea vectorului de întrerupere, citeşte din memorie de la adresa respectivă adresa primei instrucţiuni din rutina de tratare (0x6200) pe care o încarcă în PC, predând controlul rutinei de serviciu.

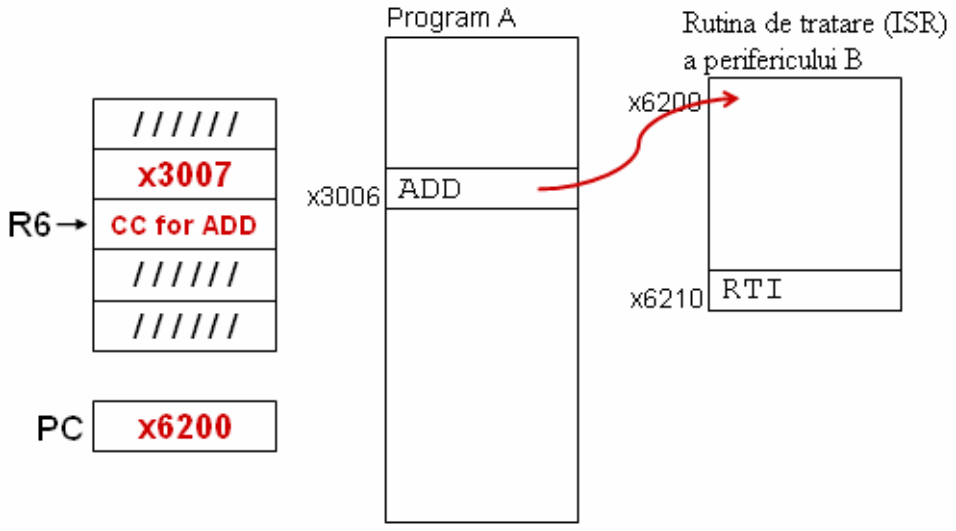


Figura 9.8. Protocolul de achitare a întreruperii – paşii 2 şi 3

Rutina de serviciu aferentă întreruperii provocate de dispozitivul B conține 17 instrucțiuni de la adresa 0x6200 la adresa 0x6210. În timpul execuției instrucțiunii AND (vezi figura 9.9) de la adresa 0x6202 dispozitivul C trimite procesorului o cerere de întrerupere.

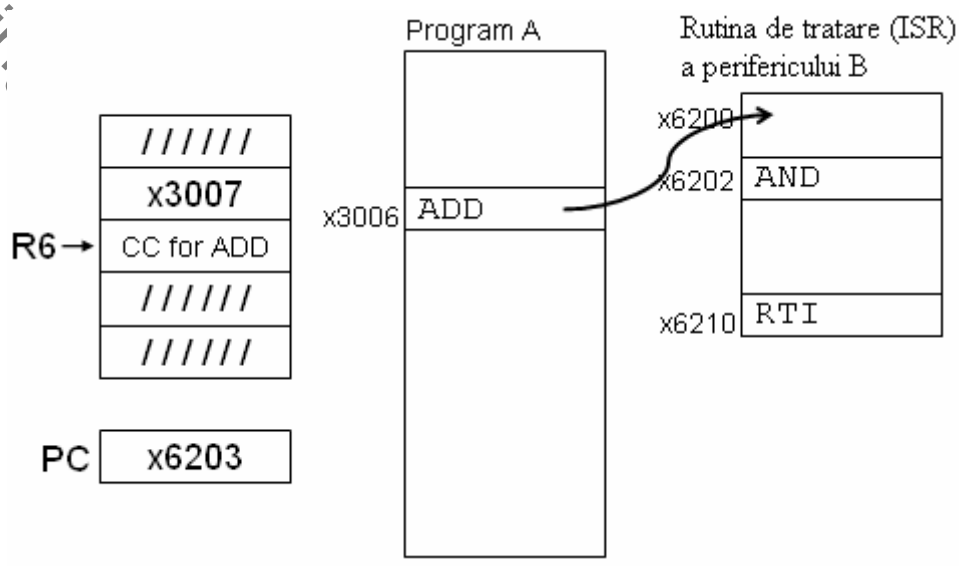


Figura 9.9. Instantaneu cu execuția instrucțiunii AND din rutina de serviciu a perifericului B

Înainte de preluarea controlului de către rutina de serviciu aferentă perifericului C (vezi figura 9.10), sunt salvate în stivă adresa instrucțiunii din rutina perifericului B, următoare celei pe care a apărut întreruperea (PC=0x6203) şi codurile de condiție (regiştrii booleeni N, Z şi P). Rutina de serviciu aferentă perifericului C este plasată la adresa 0x6300 şi conține 22 de instrucțiuni.

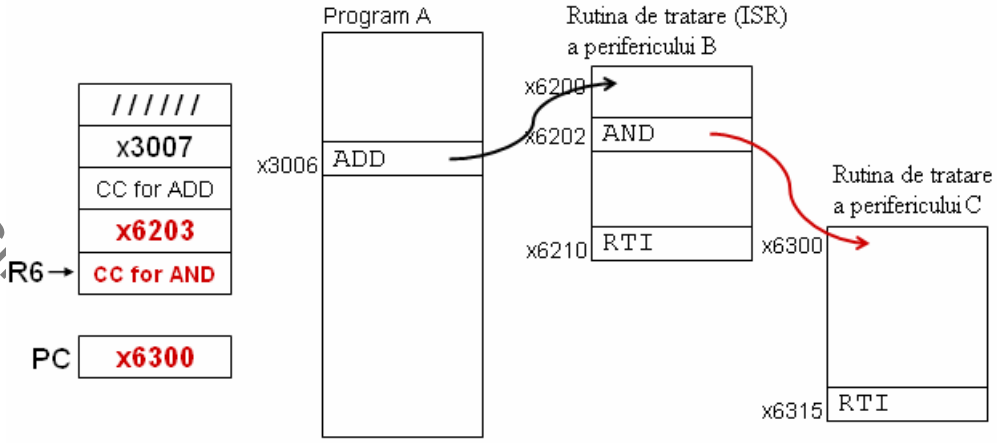


Figura 9.10. Instantaneu cu stiva, programul principal şi cele două rutine de serviciu

Execuția instrucțiunii RTI (*return from interupt*) de la adresa 0x6315 (vezi figura 9.11), ultima din rutina de tratare aferentă perifericului C, presupune extragerea din stivă atât a codurilor de condiție cât şi a adresei instrucțiunii succesoare AND-ului pe care a apărut întreruperea de la perifericul C. În continuare sunt executate instrucțiunile din rutina de serviciu aferentă perifericului B.

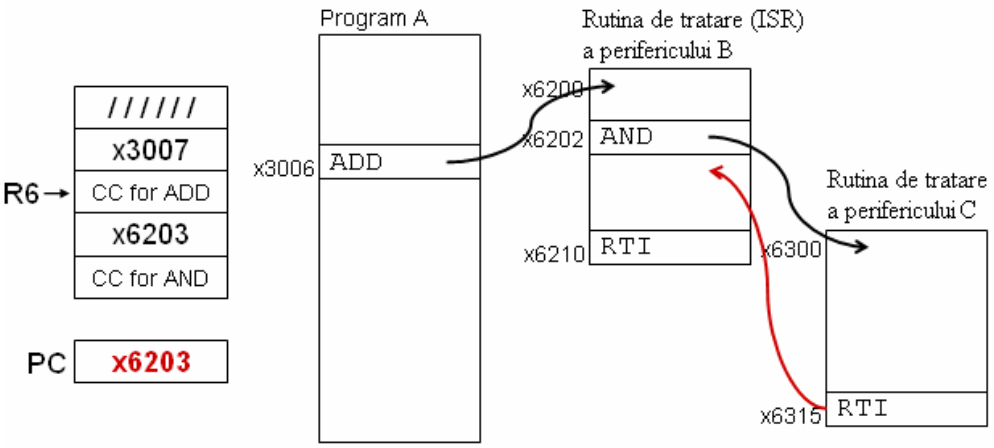


Figura 9.11. Revenirea din rutina de serviciu aferentă perifericului C

Execuția instrucțiunii RTI (*return from interupt*) de la adresa 0x6210 (vezi figura 9.12), ultima din rutina de tratare aferentă perifericului B, presupune extragerea din stivă atât a codurilor de condiție cât şi a adresei

instrucțiunii succesoare ADD-ului pe care a apărut întreruperea de la periferic B. În continuare sunt executate instrucțiunile din programul principal. În acest moment se observă cum indicatorul de vârf al stivei revine pe poziția inițială (dinaintea apariției întreruperii provocate de periferic B) când stiva era liberă.

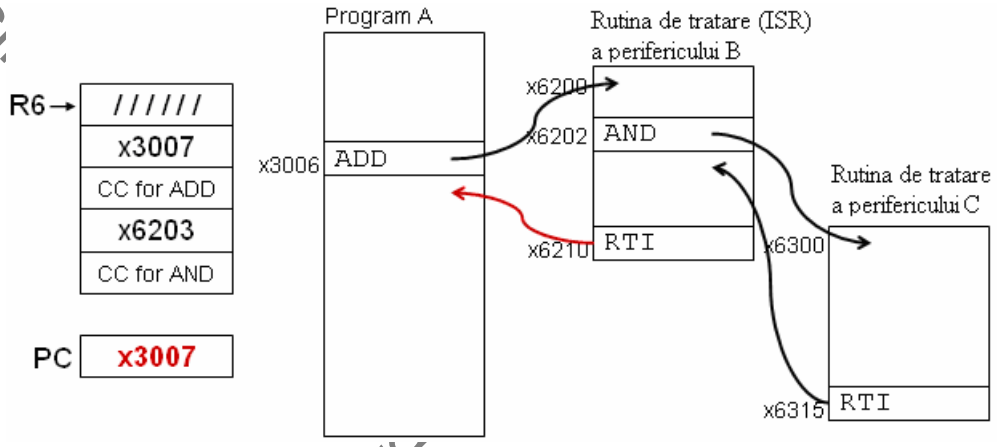


Figura 9.12. Efectul instrucțiunii RTI asupra stivei

În figura 9.13 se prezintă sintetic un instantaneu cu etapele parcurse la apariția întreruperilor provocate de cele două periferice (B și C), și care au fost detaliate prin intermediul figurilor 9.7-9.12. La momentul inițial stiva este liberă, programul principal A execută instrucțiunea de adunare de la adresa 0x3006, moment în care periferic B lansează o cerere de întrerupere. La finele fazei de *Scriere Rezultat* aferente instrucțiunii ADD (procesorul acceptă întreruperea, efectuându-se protocolul de achitare a acesteia), sunt salvate în stivă adresa instrucțiunii următoare ADD-ului din programul A și codurile de condiție, cedându-se controlul primei instrucțiuni din rutina de tratare a întreruperii provocate de periferic B (etapa marcata cu **1**). În timpul execuției instrucțiunii AND de la adresa 0x6202 din rutina de serviciu aferenta perifericului B, dispozitivul C trimite procesorului o cerere de întrerupere. La finele fazei de *Scriere Rezultat* aferente instrucțiunii AND (procesorul accepta întreruperea, efectuându-se protocolul de achitare a acesteia), sunt salvate în stivă adresa instrucțiunii următoare AND-ului din rutina de serviciu a perifericului B și codurile de condiție, cedându-se controlul primei instrucțiuni din rutina de tratare a întreruperii provocate de periferic C (etapa marcata cu **2**). Execuția instrucțiunii RTI de la adresa 0x6315, ultima din rutina de tratare aferentă perifericului C, presupune extragerea din stivă atât a codurilor de condiție cât și a adresei instrucțiunii succesoare AND-ului pe care a apărut

întreruperea de la perifericul C. Aceasta marchează practic etapa cu numărul **3**). În continuare sunt executate instrucţiunile din rutina de serviciu aferentă perifericului B. Execuţia instrucţiunii RTI de la adresa 0x6210, ultima din rutina de tratare aferentă perifericului B, presupune extragerea din stivă atât a codurilor de condiţie cât şi a adresei instrucţiunii succesoare ADD-ului pe care a apărut întreruperea de la perifericul B, marcând practic etapa cu numărul **4**. În continuare sunt executate instrucţiunile din programul principal ca si cum nimic nu s-ar fi întâmplat, cu o oarecare întârziere din punct de vedere al timpului de execuţie al programului A. În acest moment se observă cum indicatorul de vârf al stivei revine pe poziţia iniţială (dinaintea apariţiei întreruperii provocate de perifericul B) când stiva era liberă. Totuşi anumite locaţii în stivă au fost afectate, chiar dacă acum ele figurează ca libere.

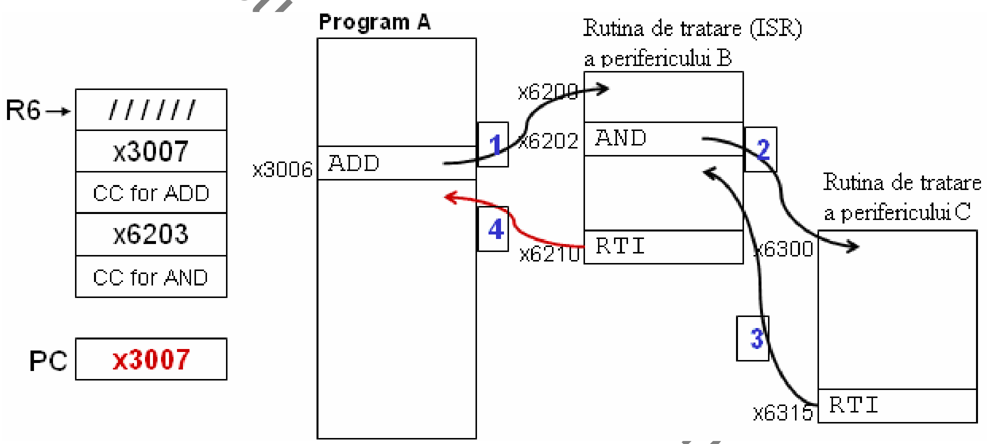


Figura 9.13. Instantaneu care sintetizează etapele parcurse la apariţia întreruperilor de la cele două periferice

9.2. OPERAŢII ARITMETICE FOLOSIND STIVA

O aplicabilitate importantă a mecanismului stivei o reprezintă implementarea maşinilor virtuale. De regulă, aceste arhitecturi folosesc în locul regiştrilor o stivă pentru operanzii sursă şi destinaţie (se mai numesc arhitecturi fără operanzi – „*zero-address machine*”). Instrucţiunile în acest caz sunt simple: *ADD*, *MUL*. Execuţia lor presupune extragerea din stivă a ultimelor două locaţii ocupate, execuţia propriu-zisă a adunării sau înmulţirii

cu cei doi operanzi aflați în stivă, rezultatul fiind apoi depus înapoi în stivă. Instrucțiunile *PUSH, POP, ADD, MUL, NEG*, etc. aparțin mașinii virtuale dar nu arhitecturii gazdă pe care rulează aceasta (în cazul de față – LC-3 ISA). Codul aferent operațiilor implicate de respectivele instrucțiuni sunt descrise prin intermediul unor rutine formate din instrucțiuni ale arhitecturii gazdă. Aceste rutine nu pot fi alterate (modificate) de către programator. Ele reprezintă de fapt semnale de control (componente microarhitecturale) folosite de către arhitectura gazdă pentru declanşarea operațiilor care au loc la nivelul mașinii virtuale.

Pentru exemplificare se consideră următoarea expresie de calculat: **E = (A+B)·(C+D)**. Dacă A reține valoarea 25, B conține valoarea 17, C este egal cu 3 şi D este egal cu 2, să se memoreze rezultatul în E.

a) *Evaluarea expresiei pe o arhitectură cu trei operanzi*:

Presupunând că LC-3 ISA pune la dispoziție o instrucțiune de înmulțire (notată cu MUL), atunci într-o arhitectură cum este LC-3 (*three-address machine*) codul sursă pentru evaluarea expresiei E = (A+B)·(C+D) este următorul:

```
LD R0, A
LD R1, B
ADD R0, R0, R1
LD R2, C
LD R3, D
ADD R2, R2, R3
MUL R0, R0, R2
ST R0, E
```

b) *Evaluarea expresiei pe o arhitectură cu zero operanzi (bazată pe stivă)*:

```
PUSH A
PUSH B
ADD
PUSH C
PUSH D
ADD
MUL
POP E
```

Un exemplu foarte des folosit în sistemele de calcul moderne îl constituie **maşina virtuală Java** (JVM). Aceasta reprezintă un model abstract de arhitectură de calcul destinată execuției unui program scris într-un limbaj independent de maşină (bytecode Java).

- ✓ Interacționează atât cu sistemul de operare cât şi cu arhitectura hardware a sistemului gazdă
- ✓ Unul din **avantajele majore** îl constituie faptul că asigură portabilitatea aplicațiilor Java (implementarea browser-elor JVM).
- ✓ Componentele de bază ale maşinii virtuale Java:
  - *Regiştri* – include un program counter (*PC*) şi trei alți regiştri folosiți pentru a administra *stiva*. Întrucât JVM este o arhitectură bazată pe stivă rezultă necesitatea unui număr redus de regiştri.
  - Bytecodul este stocat în *zona de cod*. Registrul PC indică spre următorul *bytecod* (instrucțiune) din zona de cod care va fi executat de JVM. Parametrii pentru instrucțiunile bytecod (metode), ca şi rezultatele executării instrucțiunilor bytecod (rezultatele returnate de metode) sunt stocate pe stivă.
  - *Stiva* este folosită şi pentru a menține contextul fiecărei metode invocate, numit *stack frame*. Pentru a administra stack frame-ul (contextul), se folosesc regiştrii *optop*, *frame* şi *vars*. Parametrii instrucțiunilor *bytecod* precum şi rezultatele execuției acestor instrucțiuni sunt memorate în stivă.

O altă aplicabilitate importantă a mecanismului stivei se regăseşte în **evaluarea expresiilor** (aritmetice, simbolice, relaționale) din cadrul analizei sintactice – componentă a compilatoarelor [Gol97].

Pentru evaluarea unei expresii aritmetice compilatorul o aduce la o formă bazată pe forma poloneză postfixată în care au fost desființate parantezele rotunde. Ordinea de evaluare a expresiilor compuse este determinată de:

- ✓ utilizarea parantezelor
- ✓ precedența operatorilor

Notația poloneză (sau *postfixată*) este un mod de scriere a expresiilor, în care ordinea operatorilor şi a operanzilor este schimbată față de cea dintr-o expresie uzuală.

În notația uzuală (numită şi *infix*) o expresie are forma: *operand* **operator** *operand*

În notația *postfixată* (poloneză) expresia este scrisă sub forma: *operand operand* **operator**.

De exemplu:
a + b devine în notație poloneză **a b +**
(a - b) * c devine **a b - c ***
a - b * (c + d) devine **a b c d + * -**

Avantajul notației poloneze este acela că indică ordinea corectă a evaluării operațiilor fără a utiliza paranteze. Astfel, o expresie poloneză poate fi evaluată printr-o singură parcurgere a sa. **Pentru evaluarea unei expresii în notație poloneză se poate folosi o stivă de valori reale**. Ori de câte ori întâlnim în expresie un operand, valoarea lui este introdusă în stivă. Daca întâlnim un operator atunci se extrag două valori din vârful stivei, care sunt, de fapt, cei doi operanzi, se aplică operatorul asupra valorilor extrase şi rezultatul este depus în stivă. În momentul în care s-a terminat parcurgerea expresiei, rezultatul final al expresiei este în vârful stivei (şi stiva conține doar această valoare).

9.3. CONVERSIA INFORMAȚIEI DIN FORMAT ASCII ÎN ZECIMAL

Conversia informației în sistemele de calcul este necesară pentru a facilita transmiterea acesteia dintr-o formă prietenoasă pentru utilizator spre procesare într-o formă prietenoasă pentru calculator. Conversia informației din format ASCII în format decimal trebuie făcută întrucât rutinele de intrare a informației în calculator (citirea de la tastatură sau din fişiere) preiau datele în format ASCII (caractere, nu valori numerice). De asemenea, ieşirea informației din sistem se face în acelaşi format. Spre exemplificare se consideră următorul program scris în limbaj de asamblare LC-3.

```
TRAP 0x23          ; întrerupere software pentru citirea de la
                     tastatură
ADD R1, R0, #0     ; transferă codul ASCII al caracterului
                     citit în R1
TRAP 0x23            ; întrerupere software pentru citirea de
                     la tastatură
ADD R0, R1, R0     ; adună cele două informații citite
                     (codurile ASCII)
```

```
TRAP 0x21       ; afişează caracterul al cărui cod ASCII
                  se află în R0
TRAP 0x25       ; HALT – întrerupere care marchează
                  încheierea programului
```

Dacă utilizatorul introduce succesiv valorile 2 şi apoi 3 rezultatul afişat pe ecran va fi caracterul „e”. Explicaţia este următoarea: codul ASCII al caracterului „2” este 0x32, cel al caracterului „3” este 0x33 rezultând în registrul R0 valoarea (0x32 + 0x33) = 0x65, iar caracterul cu acest cod ASCII este „e”.

9.3.1. CONVERSIA DIN ASCII ÎN BINAR

Exemplul din paragraful anterior este însă valabil doar dacă se citesc de la tastatură numere alcătuite dintr-o cifră. În continuare va fi prezentată o secvenţă de program care permite conversia unui şir de caractere într-un număr alcătuit din mai mulţi digiţi. Se consideră astfel că, de la tastatură se citeşte şirul de caractere "259" şi se stochează într-o zonă de memorie, de unde va fi luat fiecare caracter în parte, transformat în digit şi prelucrat pentru generarea numărului final. Etapele conversiei sunt următoarele:

- Se converteşte primul caracter al şirului (cel mai semnificativ – în acest caz 2) în digitul corespunzător (se scade valoarea 0x30) şi se înmulţeşte cu 100.
- Se converteşte al doilea caracter al şirului (în acest caz 5) în digitul corespunzător şi se înmulţeşte cu 10.
- Se converteşte al treilea caracter al şirului (în acest caz 9) în digitul corespunzător. Dacă numărul ar fi avut mai mulţi digiţi trebuia iniţial identificată lungimea şirului, iar aflarea fiecărei cifre continua până la cifra unităţilor.
- Se adaugă toate cele trei numere obţinute la fiecare pas.

Una din operaţiile amintite anterior şi necesară în procesul de conversie o reprezintă înmulţirea. Întrucât arhitectura LC-3 ISA nu dispune de o instrucţiune de înmulţire trebuie găsită o alternativă prin care să se poată realiza înmulţirea unui digit cu 100 şi apoi înmulţirea unui digit cu 10. O abordare simplistă ar consta în adunări repetate (se adună digitul respectiv de 100 de ori), variantă ineficientă totuşi. O a doua variantă o reprezintă adunarea numărului 100 cu el însuşi de *valoarea digitului* respectiv ori (mai eficientă decât varianta anterioară întrucât se fac mai puţine adunări – digitul fiind în mod clar <10). Chiar şi această a doua variantă se poate

realiza printr-un artificiu. Decât să se efectueze un număr necunoscut de adunări se apelează la o tabelă de căutare folosită în cazul înmulțirii. Astfel, fiecare intrare în această tabelă va conține un număr (multiplu de 100) între 0 şi 900 inclusiv, digitul reprezentând indexul în intervalul (0-9) în tabelă.

Entry 0: 0 x 100 = 0
Entry 1: 1 x 100 = 100
Entry 2: 2 x 100 = 200
etc.

Secvența de cod asamblare LC-3 pentru căutarea în tabelă este următorul:

```
; se înmulțeşte conținutul lui R0 cu 100, folosind tabela de căutare;
        LEA R1, Lookup100       ;R1 = adresa de bază a tabelei
        ADD R1, R1, R0          ;se adaugă la adresa de bază
                                indexul aflat în R0
        LDR R0, R1, #0          ; se citeşte locația de memorie
                                de la adresa calculată în R1
        ...
Lookup100   .FILL 0             ; intrarea 0 reține valoarea
                                0*100=0
        .FILL 100               ; intrarea 1 reține valoarea
                                1*100=100
        .FILL 200               ; intrarea 2 reține valoarea
                                2*100=200
        .FILL 300               ; intrarea 3 reține valoarea
                                3*100=300
        .FILL 400               ; intrarea 4 reține valoarea
                                4*100=400
        .FILL 500               ; intrarea 5 reține valoarea
                                5*100=500
        .FILL 600               ; intrarea 6 reține valoarea
                                6*100=600
        .FILL 700               ; intrarea 7 reține valoarea
                                7*100=700
        .FILL 800               ; intrarea 8 reține valoarea
                                8*100=800
        .FILL 900               ; intrarea 9 reține valoarea
                                9*100=900
```

Rutina de conversie completă din ASCII în decimal

; Buffer-ul de trei digiți se numeşte ASCIIBUF.
; R1 indică numărul de digiți care trebuie convertiți. Conversia se va face de la cifra unităților spre sute – de la cel mai puțin semnificativ spre cel semnificativ.
; Rezultatul sub forma unui număr zecimal este depus în R0.

```
ASCIItoBinary   AND R0, R0, #0      ; inițializare rezultat
                ADD R1, R1, #0      ; testează numărul de digiți
                                    rămaşi pentru conversie
                                    ; deşi pare să nu facă nimic,
                                    instrucțiunea anterioară setează
                                    bitul de condiție
                BRz DoneAtoB        ; se sare la eticheta care
                                    marchează ultima instrucțiune din
                                    rutină dacă nu mai sunt digiți de
                                    convertit (bitul Z a fost setat pe 1
                                    de adunarea anterioară)
;
                LD R3, NegZero      ; R3 = -0x30 – se pregăteşte
                                    decrementarea codului ASCII al
                                    caracterului cu inversul codului
                                    cifrei „0”
                LEA R2, ASCIIBUF    ; R2=adresa de bază a primului
                                    element din şir (ASCIIBUF)
                ADD R2, R2, R1      ; R2 indică după caracterul de
                                    convertit (în prima fază chiar
                                    ultimul)
                ADD R2, R2, #-1     ; R2 indică în fața caracterului
                                    de convertit
                LDR R4, R2, #0      ; citeşte din tabelă caracterul de
                                    convertit
                ADD R4, R4, R3      ; converteşte caracterul în digit
                ADD R0, R0, R4      ; adaugă digitul (cifra
                                    unităților) la numărul final
;
                ADD R1, R1, #-1     ; decrementează numărul de
                                    digiți care mai trebuie convertiți,
                                    cu implicații implicite şi asupra
                                    citirii din buffer-ul de caractere
```

```
                BRz DoneAtoB        ; se sare la eticheta care marchează
                                      ultima instrucțiune din rutină dacă
                                      nu mai sunt digiți de convertit
                ADD R2, R2, #-1     ; R2 indică spre digitul zecilor
                LDR R4, R2, #0      ; citeşte din tabelă caracterul de
                                      convertit
                ADD R4, R4, R3      ; converteşte caracterul în digit
                LEA R5, Lookup10    ; R5 reține adresa tabelei cu zeci
                                      (multiplii de 10)
                ADD R5, R5, R4      ; se determină ofsetul în tabela
                                      zecilor -> R5
                LDR R4, R5, #0      ; în R4 se depune multiplul de 10
                                      respectiv
                ADD R0, R0, R4      ; se adaugă la unități şi zecile
                                      aferente numărului de convertit
;
                ADD R1, R1, #-1     ; decrementează cu 1 numărul de
                                      digiți care mai trebuie convertiți
                BRz DoneAtoB        ; se sare la eticheta care marchează
                                      ultima instrucțiune din rutină dacă
                                      nu mai sunt digiți de convertit
                ADD R2, R2, #-1     ; R2 indică spre digitul sutelor
                LDR R4, R2, #0      ; citeşte din tabelă caracterul de
                                      convertit
                ADD R4, R4, R3      ; converteşte caracterul în digit
                LEA R5, Lookup100   ; R5 reține adresa tabelei cu sutele
                                      (multiplii de 100);
                ADD R5, R5, R4      ; se determină ofsetul în tabela
                                      sutelor -> R5
                LDR R4, R5, #0      ; în R4 se depune multiplul de 100
                                      respectiv
                ADD R0, R0, R4      ; se adaugă la numărul deja format
                                      şi contribuția sutelor
;
DoneAtoB    RET                     ; revenirea din rutina de conversie
NegZero     .FILL 0xFFD0            ; -0x30 – inversul codului ASCII al
                                      caracterului „0”.
ASCIIBUF    .BLKW 4                 ; se alocă 4 spații pentru şirul de
                                      caractere de trei digiți şi pentru
                                      terminatorul NULL
Lookup10        .FILL 0
```

```
                .FILL 10
                .FILL 20
                ...
Lookup100       .FILL 0
                .FILL 100
                ...
```

9.3.2. CONVERSIA DIN ZECIMAL ÎN ASCII

Întrucât rezultatul oricărui program numeric va fi una sau mai multe valori care se doresc a fi afişate, este absolut necesară şi conversia inversă din zecimal în şir de caractere. Afişarea în LC-3 se poate face apoi folosind întreruperea software PUTS – TRAP 0x22 (pentru întreg şirul) sau OUT – TRAP 0x21 (pentru un singur caracter). Se consideră spre exemplificare un rezultat pe trei digiţi. În acest caz, operaţiile aritmetice ce trebuie efectuate sunt cele de împărţire cu 100 pentru obţinerea fiecărui digit în parte şi adunarea la acesta codul ASCII al caracterului „0” – 0x30. Împărţirea în acest caz se realizează prin scăderi repetate a valorii 100 din numărul respectiv până devine strict mai mic decât 100. Numărul de scăderi reprezintă de fapt digitul căutat – corespunzător sutelor. În continuare se scade 10 din numărul rămas până se ajunge la o valoare strict mai mică decât 10. Numărul de scăderi reprezintă digitul căutat – corespunzător zecilor. Numărul rămas reprezintă chiar cifra unităţilor.

Primul pas îl reprezintă verificarea semnului numărului. Semnul este scris în buffer urmând ca, dacă numărul este negativ să fie transformat într-unul pozitiv.

Rutina de conversie din decimal în ASCII

```
; R0 reprezintă un număr întreg de trei cifre din intervalul -999 şi +999.
; În buffer-ul ASCIIBUF se înscrie pe prima poziţie semnul urmat de trei
digiţi în format ASCII.

BinaryToASCII
                LEA R1, ASCIIBUF    ; R1 – adresa şirului de caractere
                                    care reţine rezultatul
                ADD R0, R0, #0      ; se testează semnul numărului
                BRn NegSign         ; dacă este număr negativ se sare la
                                    secvenţa care scrie semnul în
```

```
                                    buffer şi transformă numărul într-
                                    unul pozitiv.
                LD R2, ASCIIplus    ; preia în R2 codul ASCII al
                                    simbolului „+”
                STR R2, R1, #0      ; scrie + pe prima poziție în buffer-
                                    ul cu rezultatul
                BR Begin100         ; numărul fiind pozitiv se trece
                                    direct la conversie
NegSign         LD R2, ASCIIneg     ; preia în R2 codul ASCII al
                                    simbolului „-”
                STR R2, R1, #0      ; scrie - pe prima poziție în buffer-
                                    ul cu rezultatul
                NOT R0, R0          ; instrucțiunea curentă şi
                                    următoarea determină
                                    complementul față de 2 al
                                    numărului aflat în R0 (transformă
                                    numărul din negativ în pozitiv)
                ADD R0, R0, #1
Begin100                            LD R2, ASCIIoffset  ; în R2
                                    se adaugă codul ASCII al
                                    caracterului „0”
                LD R3, Neg100       ; în R3 se încarcă inversul lui 100
                                    pentru scăderi repetate
Loop100         ADD R0, R0, R3      ; scad 100 din numărul original
                                    până obțin un număr mai mic decât
                                    0
                BRn End100          ; dacă rezultatul este mai mic decât
                                    0 înseamnă că am scăzut prea mult
                                    (o dată în plus)
                ADD R2, R2, #1      ; În R2 se calculează digitul de
                                    pe poziția sutelor incrementat la
                                    fiecare scădere
                BR Loop100          ; continui şi mai scad 100
End100          STR R2, R1, #1      ; memorez digitul sutelor în buffer
                                    pe poziția a doua – după semn.
                LD R3, Pos100
                ADD R0, R0, R3      ; restabilesc valoarea corectă mai
                                    mică decât 100 în R0 prin
                                    adunarea valorii 100 la rezultatul
                                    scăderii care s-a făcut o dată în
                                    plus
```

```
;
                LD R2, ASCIIoffset  ; în R2 se adaugă codul ASCII al
                                    caracterului „0”
                LD R3, Neg10        ; în R3 se încarcă inversul lui 10
                                    pentru scăderi repetate
Loop10          ADD R0, R0, R3      ; scad 10 din numărul format doar
                                    din zeci şi unități până obțin un
                                    număr mai mic decât 0
                BRn End10           ; dacă rezultatul este mai mic decât
                                    0 înseamnă că am scăzut prea mult
                                    (o dată în plus)
                ADD R2, R2, #1      ; În R2 se calculează digitul de pe
                                    poziția zecilor incrementat la
                                    fiecare scădere
                BR Loop10           ; continui şi mai scad 10
End10           STR R2, R1, #2      ; memorez digitul zecilor în buffer
                                    pe poziția a treia – după semn şi
                                    digitul sutelor
                ADD R0, R0, #10     ; restabilesc valoarea corectă mai
                                    mică decât 10 în R0 prin adunarea
                                    valorii 10 la rezultatul scăderii care
                                    s-a făcut o dată în plus
;
                LD R2, ASCIIoffset  ; în R2 se adaugă codul ASCII al
                                    caracterului „0”
                ADD R2, R2, R0      ; se convertește digitul unităților în
                                    caracter
                STR R2, R1, #3      ; memorez digitul zecilor în buffer
                                    pe poziția a patra – după semn,
                                    digitul sutelor şi după digitul
                                    zecilor
                RET                 ; revenirea din rutina de conversie
ASCIIplus           .FILL 0x2B      ; codul ASCII al simbolului „+”
ASCIIneg            .FILL 0x2D      ; codul ASCII al simbolului „-”
ASCIIoffset     .FILL 0x30          ; codul ASCII al simbolului „0”
Neg100          .FILL 0xFF9C        ; inversul valorii 100 (pentru
                                    scăderi repetate ale sutelor)
Pos100          .FILL 100
Neg10           .FILL 0xFFF6        ; inversul valorii 10 (pentru scăderi
                                    repetate ale zecilor)
```

9.4. EXERCIȚII ŞI PROBLEME

1. Care sunt caracteristicile definitorii ale stivei? Care este rolul indicatorului spre vârful stivei (*stack pointer*)? Ce operații cu stiva cunoaşteți şi cum sunt ele implementate din punct de vedere al accesului la memorie? Există anumite condiții (cazuri speciale) care trebuie tratate?

2. Se consideră un şir de intrare pe stivă o listă de elemente care sunt încărcate pe stivă în ordinea stabilită de program (programator). De exemplu, pentru următoarele operații efectuate la nivelul stivei:

PUSH A, PUSH B, POP, PUSH C, PUSH D, POP, PUSH E, POP, POP, PUSH F, PUSH G, PUSH H, PUSH I, PUSH J, POP, PUSH K, POP, POP, POP, PUSH L, POP, POP, PUSH M

şirul de intrare este următorul: ABCDEFGHIJKLM. Şirul de ieşire reprezintă totalitatea elementelor care se extrag din stivă în ordinea stabilită de program (programator).

a) Care este şirul de ieşire pentru şirul de intrare specificat anterior? Indicație: BD...

b) Dacă şirul de intrare este ZYXWVUTSR, creați o secvență de operații cu stiva (push şi respectiv pop) astfel încât şirul de ieşire să fie YXVUWZSRT.

3. Se reaminteşte că o arhitectură fără operanzi (zero-address machine) reprezintă o arhitectură bazată pe stivă în care toate operațiile aritmetico-logice sunt realizate folosind valorile din vârful stivei. Pentru această problemă, se presupune că ISA proprie permite următoarele operații:

- **PUSH *M*** – depune în vârful stivei valoarea stocată la adresa de memorie M.
- **POP *M*** – extrage valoarea aflată în vârful stivei şi o depune în memorie la adresa M.
- **OP** – Extrage primele două valori aflate în vârful stivei şi execută operația aritmetico-logică specificată asupra celor două valori. Rezultatul este încărcat înapoi pe stivă.

Pentru această problemă opcode-urile (OP) necesare sunt: ADD şi MUL.

a) Scrieți în limbaj de asamblare secvența de cod care calculează expresia:

$$x = (A * ((B * C) + D))$$

- ✓ Într-o arhitectură fără operanzi de genul celei anterior specificate
- ✓ Într-o arhitectură cu trei operanzi – de genul LC-3 ISA, dar care încorporează suplimentar şi instrucțiunea de înmulțire – MUL.

b) Specificați câte un avantaj şi respectiv un dezavantaj al fiecărui tip de mașină din cele anterior descrise.

4. Se consideră cele două operații de depunere în stivă – Push şi respectiv de extragere din stivă –Pop. Operația Push *Rn* depune în vârful stivei valoarea conținută în registrul specificat (*Rn*). Operația Pop *Rn* extrage valoarea din vârful stivei şi o depune în registrul specificat *Rn*. Mai jos este descris un instantaneu cu cei opt regiştrii ai arhitecturii LC-3 înainte şi după execuția a şase operații cu stiva. Trebuie menționat că patru din şase operații nu sunt complet specificate. Completați în locul celor patru linii goale cu numele regiştrilor corespunzători pentru ca instantaneul să fie corect.

```
PUSH    R4
PUSH    __
POP     __
PUSH    __
POP     R2
POP     __
```

	ÎNAINTE de execuție	**DUPĂ de execuție**
R0	0x0000	0x1111
R1	0x1111	0x1111
R2	0x2222	0x3333
R3	0x3333	0x3333
R4	0x4444	0x4444
R5	0x5555	0x5555
R6	0x6666	0x6666
R7	0x7777	0x4444

5. Se consideră următoarele operații realizate la nivelul stivei:

PUSH A, PUSH B, POP, PUSH C, PUSH D, POP, PUSH E, POP, POP, PUSH F

a. Care este conținutul stivei după ultima depunere – **PUSH F**?
b. În care moment de timp stiva conține numărul maxim de elemente?
c. Fără a elimina vreunul din elementele rămase pe stivă de pe urma operațiilor anterioare, care va fi conținutul stivei după execuția următoarelor operații?

PUSH G, PUSH H, PUSH I, PUSH J, POP, PUSH K, POP, POP, POP, PUSH L, POP, POP, PUSH M

6. Răspundeți la întrebările:

a) Care sunt cele mai uzuale operații cu stiva şi care este modul lor de lucru ? Restricții ce pot să apară.
b) Cum este utilizată stiva în mecanismul de întreruperi hardware ?

7. Cu ajutorul structurilor de date dinamice implementați următoarele operații asupra unei stive de date având elemente numere întregi:

- Creare_stiva(); - creează o stivă nouă (vidă).
- Empty(stiva); – testează dacă stiva primită ca parametru este vidă, returnând 1 în caz afirmativ şi 0 dacă stiva nu este vidă.
- Push(stiva, element); – depune elementul specificat ca al doilea parametru pe stiva specificată de către primul parametru.
- Pop(stiva); – extrage şi returnează elementul din vârful stivei primită ca parametru.
- Top(stiva); – permite accesul la elementul din vârful stivei fără însă a-l *elimina* din stivă. Returnează respectivul element.
- Full(stiva); – testează dacă stiva este plină returnând 1 în caz afirmativ şi 0 în caz contrar.

Atât funcțiile *Pop* cât şi *Top* necesită testarea condiției ca stiva primită ca parametru să nu fie vidă.

8. Implementați o maşină virtuală bazată pe stivă (*zero-address machine*), folosind ca şi arhitectură de bază (gazdă) LC-3 ISA: instrucțiuni,

regiştrii – cu nume şi funcţiile stabilite (R6 – *stack pointer, etc.*). Instrucţiunile maşinii virtuale vor fi: *OpAdd*, *OpMult*, *OpNeg*, *Push M*, *Pop M*. Implementaţi toate aceste instrucţiuni şi rulaţi programele din subcapitolul 9.2 şi evaluaţi timpul de execuţie (eventual număr de instrucţiuni aferente arhitecturii gazdă executate) în ambele situaţii.

9. Implementaţi într-un program de nivel înalt (C, C++, Java) următoarea cerinţă: Se citeşte dintr-un fişier o expresie aritmetică în care operanzii sunt simbolizaţi prin litere mici (de la „a” la „z”) iar operatorii sunt „+”,„-”,„*”,„/” cu semnificaţia cunoscută. Se pot folosi şi parantezele rotunde: „(” şi „)”. Programul trebuie să verifice dacă expresia este corectă din punct de vedere sintactic (*parantezele trebuie să se închidă corect, să nu existe doi operatori sau operanzi unul după celălalt, expresia să nu se încheie cu un operator*). Se va ţine cont de următoarele indicaţii: o expresie (S) este alcătuită din termeni (T), separaţi prin „+” sau „-”; un termen este alcătuit din factori (F) separaţi prin „*” sau „/”; un factor este sau o literă mică sau o expresie cuprinsă între paranteze.

10. Implementaţi într-un program de nivel înalt (C, C++, Java) următoarea problemă: Se citeşte dintr-un fişier o expresie aritmetică în care operanzii sunt simbolizaţi prin litere mici (de la „a” la „z”) iar operatorii sunt „+”,„-”,„*”,„/”,„(” şi „)” cu semnificaţia cunoscută. Se cere să se convertească această expresie într-o expresie în forma poloneză (postfixată) – vezi subcapitolul 9.2.

11. Implementaţi într-un program de nivel înalt (C, C++, Java) următoarea problemă: Dându-se o expresie în forma poloneză (postfixată), realizaţi programul care calculează expresia utilizând numai instrucţiuni de atribuire cu un singur operator. Se vor introduce variabilele auxiliare x_1, x_2, …, x_n, unde n reprezintă numărul operaţiilor care apar în expresie. Să se verifice aplicaţia pentru diverse expresii şi pentru diverse valori aferente fiecărui operand (simbol).

10. FACILITĂȚI ALE MAŞINII PENTRU IMPLEMENTAREA ÎN HARDWARE A FUNCȚIILOR DIN PROGRAMELE DE NIVEL ÎNALT. STIVA DE DATE AFERENTĂ FUNCȚIILOR

10.1. OBȚINEREA CODULUI OBIECT PENTRU O ARHITECTURĂ DATĂ

Limbajele de programare de nivel înalt sunt independente de maşina (microarhitectura hardware) pe care se procesează. Nivelul hardware este abstractizat din punct de vedere al limbajului (operațiile nu depind de arhitectura setului de instrucțiuni – ISA) [Patt03, Zah04]. *C* este primul limbaj de nivel mediu (considerat de nivel înalt de către alți cercetători) destinat creării de sisteme de operare (anterior acestuia sistemele de operare erau scrise în limbaje de asamblare). De asemenea, în C este permisă manipularea structurilor hardware ale calculatoarelor (regiştrii, memorie, porturi). C este un limbaj de programare standardizat, compilat, implementat pe marea majoritate a platformelor de calcul existente azi. Este apreciat pentru eficiența codului obiect pe care îl poate genera, şi pentru portabilitatea sa. A fost dezvoltat la începutul anilor 1970 de Brian Kernighan şi Dennis Ritchie, care aveau nevoie de un limbaj simplu şi portabil pentru scrierea nucleului sistemului de operare UNIX. Sintaxa limbajului C a stat la baza multor limbaje create ulterior şi încă populare azi: C++, Java, JavaScript, C#. *C* este un limbaj *case sensitive*[14].

Instrucțiunile din C care rezolvă o problemă sunt mult mai puține decât cele ale limbajului asamblare aferent calculatorului LC-3 care rezolvă aceeaşi problemă [Patt03]. De asemenea, LC-3 nu asigură instrucțiuni de înmulțire şi împărțire. Limbajele de nivel înalt sunt mai expresive decât cele de nivel mediu sau jos (asamblare). Sunt folosite simboluri pentru variabile

[14] Case sensitive – literele mari sunt diferite de cele mici în cadrul tuturor construcțiilor sintactice: variabile, expresii, etc.

şi expresii simple pentru structuri de control (if-else, switch-case) sau repetitive (for, while, do-while). Permite compilarea condiționată. Deşi limbajul C oferă o libertate foarte mare în scrierea codului, acceptând multe forme de scriere care în alte limbaje nu sunt permise, acest lucru poate fi şi un dezavantaj, în special pentru programatorii fără experiență.

Compilare vs. Interpretare

Programele de nivel înalt (*High Level Languages* – HLL) pot fi translatate în instrucțiuni maşină (ISA corespunzătoare arhitecturii hardware pe care se va procesa) prin două metode [Aho86, Gol97]:

- ✓ **Interpretare:**
  Codul sursă HLL este „*interpretat*" de o maşină virtuală (cele mai cunoscute sunt maşinile virtuale Java – JVM) care translatează „*secțiuni*" din codul sursă în limbaj maşină şi îl execută direct pe arhitectura gazdă, trecând apoi la următoarea *secțiune* de cod sursă. Dintre avantajele interpretării față de metodele tradiționale de compilare s-ar menționa simplitatea implementării hardware dar şi faptul că nu necesită o zonă mare de memorie pentru stocarea programului compilat. Principalul dezavantaj îl reprezintă viteza scăzută de execuție a aplicației, dar şi necesitatea existenței în momentul interpretării atât a codului sursă HLL cât şi a maşinii virtuale care realizează interpretarea.
- ✓ **Compilare:**
  Compilarea directă translatează codul sursă (poate fi Java, C, C++, Fortran, Pascal) în instrucțiuni maşină, direct executabile pe un procesor țintă, dar nu le execută ca în cazul interpretoarelor. Orice modificare a codului sursă necesită recompilare. Procesul este realizat static şi poate îngloba tehnici de optimizare de tip analiza dependențelor de date dintre instrucțiuni, analiză interprocedurală. Se caracterizează printr-o lipsă de portabilitate.

10.1.1. DESCRIEREA COMPONENTELOR UNUI COMPILATOR

Compilatorul translatează un program sursă scris într-un limbaj de nivel înalt (C, Pascal) în acelaşi program - obiect, de regulă scris în limbaj de asamblare. În literatura de specialitate [Aho86, Gol97], este denumită *fază* a unui compilator o succesiune de operațiuni prin care un program de la

intrare suferă anumite modificări. Prin *trecere* aparţinând unui compilator se înţelege o citire a programului dintr-un fişier, transformarea lui conform unor faze şi scrierea rezultatului în alt fişier(de ieşire).

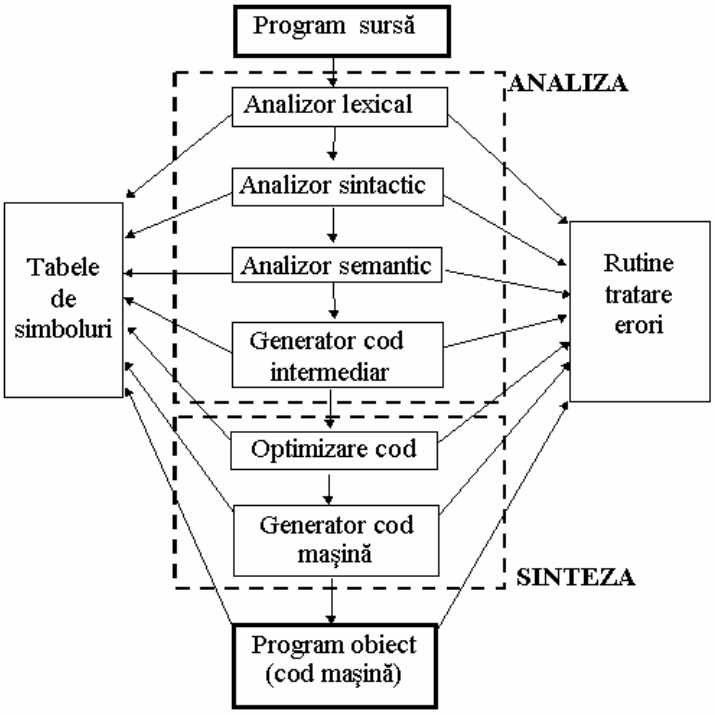


Figura 10.1. Fazele procesului de compilare

Figura anterioară (10.1) ilustrează cele două faze majore ale procesului de compilare propriu-zisă:

- **analiza** (de tip "*front end*") - în care se identifică părţile constituente fundamentale ale programului (variabile, expresii, declaraţii, definiţii, apeluri de funcţii) şi se construieşte o reprezentare internă a programului original, numită „*cod intermediar*" (*analiza lexicală* produce un -> şir de atomi lexicali -> *analiza sintactică* generează -> arborele sintactic -> *analiză semantică* construieşte o reprezentare a programului sursă în-> *cod intermediar*). Este dependentă de limbajul de nivel înalt şi nu de maşina pe care se va procesa.
- **sinteza** (de tip "*back end*") - generează cod maşină eventual optimizat. Se disting două etape:
  - optimizare cod intermediar (scheduling) pentru o anumită maşină;
  - generare de cod maşină (generare cod într-o gamă variată de formate: *limbaj maşină absolut*, *limbaj maşină relocabil* sau

limbaj de asamblare urmat de alocare de resurse). Această fază este puternic dependentă de maşina pe care se va executa codul obiect generat.

Înaintea primei faze din cadrul procesului de compilare trebuie realizată de cele mai multe ori o **preprocesare**. Preprocesorul translatează un program al cărui limbaj sursă este de nivel înalt (C, Pascal) într-un program destinaţie (obiect) scris tot într-un limbaj de nivel înalt C. Practic are loc interpretarea *directivelor de preprocesare* (încep cu # - *include, define , if !defined*(**…**)**…***endif*). De exemplu, preprocesorul trebuie să introducă conţinutul fişierului <*stdio.h*> în codul sursă al aplicaţiei create acolo unde apare respectiva directivă de preprocesare, sau, să înlocuiască anumite valori constante declarate ca şiruri de caractere cu valorile numerice aferente, dacă acest lucru se specifică printr-o directivă de tip *#define* în cadrul programului.

Analiza lexicală reprezintă prima fază a procesului de compilare, şi care este responsabilă de transformarea programului sursă văzut ca o succesiune de caractere (text) într-o succesiune de atomi lexicali şi de atribute ale lor. Conform definiţiei din [Gol97], un atom lexical este o entitate indivizibilă a unui program - identificatori de variabile/funcţii, constante de diverse tipuri, operatori, cuvinte cheie (if, else, switch/case…), delimitatori. Atributele atomilor lexicali sunt: clasă, tip, lungime, loc în tabela de simboluri, dacă a fost definit sau nu, etc. Clasa atributului se referă la faptul că este cuvânt-cheie, identificator de variabilă, constantă de tip întreg sau real etc.

În continuare sunt descrise câteva consideretne care demonstrează utilitatea analizei lexicale ca fază separată a compilatorului:

1. Asigură o proiectare mai uşoară a compilatorului.
2. Faza necesită operaţii de nivel fizic: citiri de fişiere disc(programul sursă), comparări de şiruri de caractere, acces la nivel de bit , caută în tabele consumând mult timp în raport cu celelalte faze. Este eficientă scrierea în limbaj de asamblare sau limbaj C a acestei faze urmând ca fazele următoare să poată fi scrise în orice limbaj de nivel înalt.
3. Procesul de analiză sintactică şi semantică este simplificat textul fiind curăţat de spaţii (blank-uri) şi comentarii.
4. Asigurarea portabilităţii algoritmului de compilare pentru diverse versiuni ale aceluiaşi limbaj de programare.

Analizorul lexical (AL) constituie interfaţa dintre programul sursă şi compilator (analizorul sintactic). Relaţia analizorului lexical (AL) faţă de analizorul sintactic (AS - cel care determină faptul că atomii lexicali aparţin sau nu limbajului de programare) este de două feluri:

a) **independent** - AL este independent de AS. Analizorul lexical primeşte la intrare textul cu programul sursă şi furnizează la ieşire un fişier sau o zonă de memorie cu succesiunea de atomi lexicali ce vor fi daţi spre prelucrare analizorului sintactic (vezi 10.2).

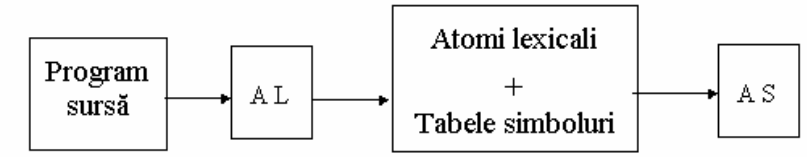


Figura 10.2. Analizorul Lexical independent de Analizorul Sintactic (1)

b) **comandat** - AL este lansat de către AS de câte ori are nevoie de un nou atom lexical (în compilatoare sub sistemul de operare DOS) sau AS şi AL sunt corutine (două procese independente care se lansează reciproc - în sisteme multitasking UNIX, Windows).

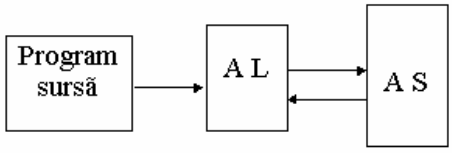


Figura 10.3. Analizorul Lexical comandat de Analizorul Sintactic

Analiza sintactică grupează atomii lexicali generaţi de AL în structuri sintactice precum: expresii, liste, instrucţiuni, proceduri. Toate acestea le plasează într-un *arbore sintactic*-cu ajutorul căruia sunt descrise relaţiile de incluziune ale structurilor sintactice, furnizând în final informaţii privitoare la corectitudinea acestora.

Analiza semantică - (componenta compilatoarelor cel mai puţin dezvoltată teoretic), face delimitările din punct de vedere semantic (al înţelesului programului). Extrage informaţiile privind apariţiile obiectelor purtătoare de date din program (variabile, tipuri de date, funcţii, etc) pentru verificarea consistenţei utilizării lor. Pe măsura parcurgerii arborelui, analiza semantică construieşte o reprezentare a programului sursă în **cod intermediar** (şir de instrucţiuni simple cu format fix, foloseau operanzi variabile de program şi nu regiştri), reflectând structura programului sursă şi înglobând toate informaţiile necesare fazelor următoare : *optimizarea* şi *generarea de cod obiect*. Generarea codului intermediar fiind legată de structura semantică se poate trata în cadrul analizei semantice.

Rezolvarea sarcinilor analizei semantice necesită de obicei mai multe treceri prin arborele de derivare ceea ce lungeşte corespunzător durata

compilării. Numărul acestor treceri poate fi fix ori, în cazul unor limbaje mai complexe sau a celor defectuos proiectate, variabil în funcţie de structura programului analizat.

Toate fazele unui compilator dispun de **rutine de tratare a erorilor** şi lucrează în comun cu una sau mai multe tabele de simboluri în care se păstrează informaţii despre cuvintele cheie ale limbajului, identificatorii de variabile (tip, lungime, adresă stocare în memorie etc.), etichete de instrucţiuni, identificatori de proceduri şi funcţii şi alte informaţii utile despre programul în curs de compilare. **Tabela de simboluri** este creată în faza de analiză a programului sursă şi folosită în scopul validării numelor simbolice facilitând generarea de cod [Gol97]. Tabela de simboluri realizează o asociere simbolică între numele (identificatorul) unei variabile şi caracteristicile acesteia (tip, domeniu de vizibilitate, deplasament faţă de începutul zonei de date statice sau dinamice, stivă). Compilatorul foloseşte tabela de simboluri pentru a urmări domeniul de utilizare şi valabilitate a unui nume şi de a adăuga informaţii despre un nume. Tabela de simboluri este cercetată de fiecare dată când un nume este întâlnit în textul sursă. Mecanismul asociat tabelei de simboluri trebuie să permită adăugarea a noi intrări şi găsirea informaţiilor existente în mod eficient. Tabela de simboluri este similară celei generate de asamblor însă cuprinde mai multe informaţii decât aceasta. În asamblare, toţi identificatorii (variabilele) erau etichete (*labels*) şi informaţiile reprezentate de acestea erau adrese. Tabela de simboluri conţine informaţii legate de toate variabilele din program.

Optimizarea de cod intermediar (**scheduling**) [Vin00, Gol97], deşi opţională, este extrem de importantă din punct de vedere al creşterii performanţei în domeniul paralelismului la nivelul instrucţiunilor. Optimizarea poate fi tratată din două perspective: **locală** (în cadrul unităţilor secvenţiale de program – basic-block-uri) şi respectiv **globală** (a întregului program). Noţiunea de *optimizare* este oarecum forţată întrucât nu există garanţia că acel cod rezultat este optim, măsurat cu vreo metrică (formulă) matematică.

Optimizarea locală a basic-block-urilor se realizează folosind algoritmi cvasioptimali într-o singură trecere, de tip "*List Scheduling*". Deoarece majoritatea programelor HLL sunt scrise în limbaje imperative şi pentru maşini secvenţiale cu un număr limitat de registre în vederea stocării temporare a variabilelor, este de aşteptat ca gradul de dependenţe între instrucţiunile adiacente să fie ridicat (paralelism relativ scăzut (2-3 instrucţiuni) la nivelul basic-block-urilor). Aşadar pentru creşterea nivelului de paralelism este necesară suprapunerea execuţiei unor instrucţiuni situate în basic-block-uri diferite (*software pipelining*, *loop unrolling*), ceea ce conduce la ideea optimizării globale.

Optimizarea globală a programului, de natură NP - completă, şi care la ora actuală constituie încă o problemă deschisă, se bazează pe algoritmi determinişti de tip "*Trace Scheduling*" sau pe algoritmi euristici de tip "*Enhanced Percolation*", "*inlining*" selectiv aplicat procedurilor [Vin00].

Dependenţele cauzate de variabilele aflate în memorie reprezintă de asemenea o frână în calea creşterii performanţei arhitecturilor de calcul. Procesarea «*Out of Order*» a instrucţiunilor cu referire la memorie într-un program este dificilă datorită accesării aceleiaşi adrese de memorie de către o instrucţiune care citeşte respectiv una care scrie. Există motive însă, ca o instrucţiune *Load* amplasată după o instrucţiune *Store* să se execute înaintea acesteia din motive de eficienţă a execuţiei (mascare latenţă, reducerea necesarului de lărgime de bandă a memoriei prin **bypassing**). Acest lucru este posibil numai dacă cele 2 adrese de memorie sunt întotdeauna diferite. Este evident că dacă la un anumit moment ele sunt identice, semantica secvenţei se modifică inacceptabil. Atunci când acest lucru este posibil, problema se rezolvă static, de către compilator. Rutina de **analiză anti-alias (***disambiguisation routine***)** [Vin00], componentă a compilatorului, compară cele 2 adrese de memorie şi returnează una dintre următoarele 3 posibilităţi:

- adrese întotdeauna distincte;
- adrese întotdeauna identice;
- cel puţin 2 adrese identice sau nu se poate determina.

Doar în primul caz putem fi siguri că execuţia anterioară a instrucţiunii Load faţă de instrucţiunea Store (sau simultană în cazul procesoarelor cu execuţie multiplă) îmbunătăţeşte performanţa fără a cauza alterarea semantică a programului. Din păcate, nu se poate decide întotdeauna acest lucru în momentul compilării.

Analiza anti-alias statică dă rezultate bune în cazul unor adresări liniare şi predictibile ale memoriei (accesări de tablouri uni sau bi-dimensionale). Prin urmare un reorganizator de program bazat pe analiza anti-alias statică va fi deosebit de conservativ în acţiunile sale. Când compararea adreselor de memorie se face pe parcursul procesării programului prin hardware, se realizează o dezambiguizare dinamică. Aceasta este mai performantă decât cea statică dar necesită resurse hardware suplimentare şi implicit costuri sporite.

Pe lângă dezambiguizarea statică mai există şi alte transformări de cod în vederea optimizării [Gol97]:

- Eliminarea instrucţiunilor redundante sau a celor de neatins (*unreachable code*) din program. De exemplu, o instrucţiune neetichetată care urmează imediat după un salt necondiţionat poate fi eliminată.

- Optimizări ale fluxului de control. De exemplu, cazul instrucţiunilor de salt condiţionate sau nu, care au ca destinaţie tot o instrucţiune de salt, sau salt condiţionat.
- Simplificări algebrice.
- Reduceri în forţă. De exemplu, înmulţirea sau împărţirea întregilor (în virgulă fixă) printr-o putere a lui 2 este mai ieftin de implementat ca o deplasare (shift) de biţi. Împărţirea în virgulă flotantă printr-o constantă poate fi implementată (aproximată) ca şi multiplicare a constantei, ceea ce poate fi ieftin.

Generatorul de cod constituie faza finală a unui compilator. Primeşte la intrare reprezentarea intermediară a programului sursă împreună cu informaţia din tabela de simboluri - folosită la determinarea run-time a adresei obiectelor de date, desemnate de nume în reprezentarea intermediară, şi produce la ieşire un program obiect echivalent. Codul de ieşire trebuie să fie corect şi de înaltă calitate, aceasta însemnând că el trebuie să folosească eficient resursele maşinii pentru a putea fi procesat cât mai rapid.

Ieşirile generatorului de cod se găsesc într-o gamă variată de forme: *limbaj de asamblare*, *limbaj maşină relocabil* sau *limbaj maşină absolut*. Producând un program într-un limbaj de asamblare, caracterizat de o mai mare lizibilitate, generarea codului se realizează mult mai uşor. Pot fi generate instrucţiuni simbolice precum şi utilizate facilităţile de macro-asamblare ajutătoare la generarea de cod. Producând un program în limbaj maşină absolut, există avantajul că programul poate fi plasat într-o locaţie fixă a memoriei şi poate fi imediat executat. Producând un program în limbaj maşină relocabil, acesta permite subprogramelor sa fie compilate separat (similar cu fişierul *Makefile* din cadrul setului de instrumente SimpleScalar 3.0 [Flo05]). O mulţime de module-obiect relocate pot fi *legate* (*linked*) împreună şi apoi încărcate pentru execuţie de către un *loader*. Câştigul de flexibilitate care dă posibilitatea compilării separate a subrutinelor şi apelarea altor programe compilate anterior dintr-un modul obiect este tributar costurilor suplimentare pentru legare şi încărcare.

Dintre problemele inerente care apar la generatoarele de cod se amintesc: managementul de memorie (mai ales în cazul instrucţiunilor de salt condiţionat), selecţia instrucţiunilor maşină, alocarea registrelor şi ordinea de evaluare.

Natura setului de instrucţiuni al maşinii destinaţie (viteza instrucţiunilor şi limbajul maşinii) determină dificultatea în selectarea instrucţiunilor ce vor fi folosite pentru generarea de cod obiect. Uniformitatea şi completitudinea setului de instrucţiuni constituie de asemenea, factori importanţi. Dacă maşina destinaţie nu suportă toate

tipurile de date într-o maniera uniformă, atunci fiecare excepţie are nevoie de o tratare deosebită.

Este cunoscut faptul că instrucţiunile cu operanzii registru (la nivelul procesorului) sunt de obicei mai scurte şi mai rapide decât cele cu operanzii în memorie. Astfel, utilizarea eficientă a registrelor este importantă în generarea de cod corect. Utilizarea registrelor este adesea subdivizată în două subprobleme [Pat05, Gol97]:

1. În timpul generării de cod, într-un anumit punct al programului, sunt alocate registrele, adică selectată mulţimea de variabile care va fi rezidentă în regiştrii.
2. Din acel punct, în timpul următoarei faze de asignare a regiştrilor(la variabile) se va selecta registrul specific ce conţine variabila rezidentă în el.

Găsirea unei asignări optime a regiştrilor la variabile este dificilă, hardware-ul şi software-ul maşinii destinaţie putând solicita anumite convenţii în utilizarea anumitor regiştrii - folosirea specializată a unora dintre aceştia. De exemplu, registrul $29 reprezintă *stack pointer*-ul, $31 - constituie adresa de revenire, sau regiştrii flotanţi dublă precizie sunt alcătuiţi din regiştrii par:impar flotanţi simplă precizie, în cazul procesorului MIPS.

De asemenea, ordinea în care sunt efectuate calculele poate afecta eficienţa codului generat, unele calcule necesitând mai puţine registre pentru rezultate intermediare, faţă de altele. Cu toate aceste probleme, compilatoarele sunt astfel proiectate încât să respecte principiile de calitate (corectitudine) şi eficienţă.

În finalul acestui paragraf este exemplificat modul de alocare al spaţiului de memorie (pentru variabile) de către compilator în cazul calculatorului LC-3. Zona de memorie utilizator (vezi figura 10.4) este compusă din trei mari regiuni:

- Instrucţiunile (zona de *cod*) la care registrul PC (*program counter*) indică spre adresa următoarei instrucţiuni de executat la fiecare moment de timp.
- Zona de *date globală* – unde sunt stocate toate variabilele declarate global în codul sursă. R5 (similar cu registrul $gp de la procesorul MIPS) va indica spre începutul acestei zone, cunoscută şi sub numele de zonă de date statice. Această zonă se caracterizează prin faptul că, în momentul compilării, se cunoaşte numele, adresa şi spaţiul ocupat de fiecare variabilă declarată aici (static).
- Stiva de date (aferentă fiecărei funcţii din programul sursă HLL). Aici sunt stocate variabilele locale, parametrii funcţiilor. R6 (similar cu

registrul $sp de la MIPS) indică spre vârful stivei. Este o zonă de date dinamică, alocată la fiecare apel de funcție și care este dealocată în momentul ieşirii din respectiva funcție. În subcapitolul 10.2 se prezintă cum este implementat în hardware mecanismul de gestiune a stivelor de date asociate funcțiilor din programele *C*.

Un singur lucru ar mai fi de menționat aici şi anume că variabilele globale sunt salvate în zona de date statice începând cu deplasamentul (*offset*) 0, iar în stiva de date aferentă fiecărei funcții variabilele şi ceilalți parametrii (dacă există) sunt salvați începând cu offset-ul 3. Preluarea conținutului variabilelor stocate poate fi făcută cu instrucțiunile de citire din memorie de tip LDR, ca în exemplul de mai jos.

- Pentru prima variabilă globală din program: LDR R1, R5, #0
- Pentru primul parametru al unei funcții: LDR R2, R6, #3

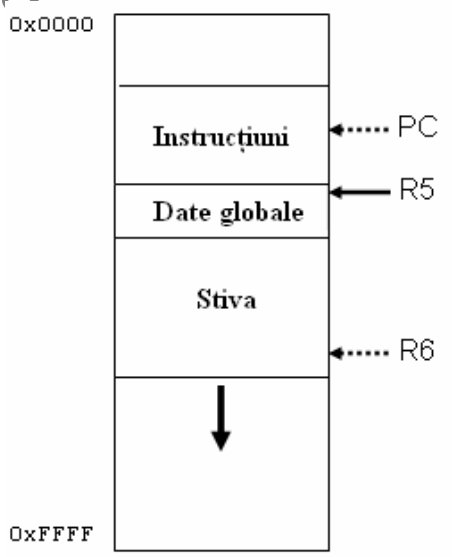


Figura 10.4. Organizarea memoriei la calculatorul LC-3

10.1.2. ETAPELE PARCURSE PENTRU RECOMPILAREA INSTRUMENTELOR *SIMPLESCALAR 3.0* [Flo05]

Creşterea în performanță şi complexitate a microprocesoarelor moderne datorate tehnicilor avansate gen *pipelining*, execuție *out-of-order* , predicție şi execuție speculativă, presupune un efort suplimentar de

proiectare şi verificare pentru dezvoltarea şi implementarea de produse viabile. Pentru depăşirea acestor probleme, proiectanţii de microarhitecturi au explorat diverse modalităţi de *transfer de funcţionalitate* la nivelul compilatorului. Începând cu procesoarele RISC[15] VLIW[16] şi continuând cu cele EPIC[17] – versiunile 1 şi 2 de procesoare Intel Itanium, compilatorul a jucat un rol important în simplificarea arhitecturii la nivel hardware menţinând totodată tendinţele curente de creştere a performanţei. În acest paragraf sunt prezentate fazele principale care trebuie parcurse pentru compilarea şi execuţia programelor de test scrise în C (sau C++) sub sistemul de operare Linux folosind utilitarele GNU. Etapele parcurse pornind de la sursa HLL a programelor de test şi până la simularea de tip *execution driven* sunt evidenţiate în figura următoare.

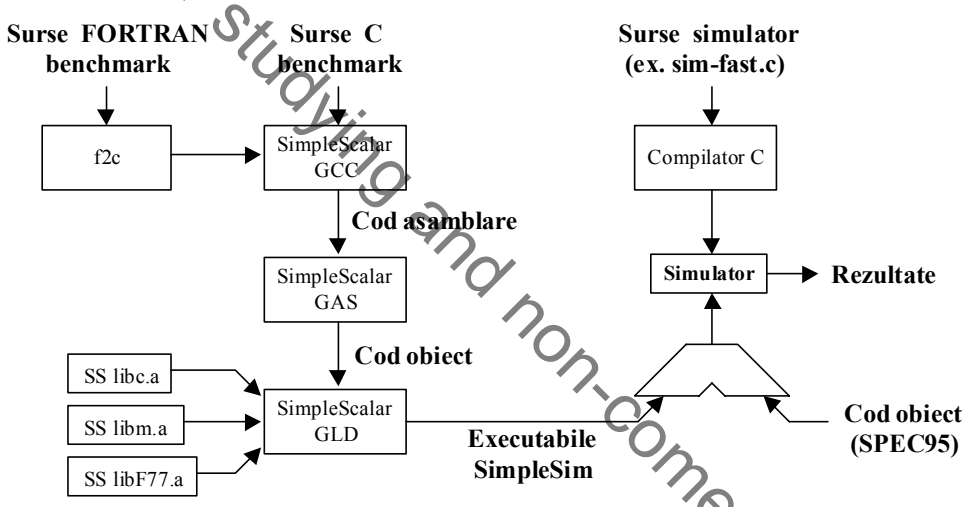


Figura 10.5. Interacţiunea instrumentelor în cadrul setului SimpleScalar

[15] RISC – calculatoare cu set redus (optimizat) de instrucţiuni

[16] VLIW – procesoare cu execuţie multiplă (Very Long Instruction Word). Din punct de vedere hardware un procesor VLIW este foarte simplu şi constă dintr-o colecţie de unităţi funcţionale de execuţie (sumatoare, circuite multiplicative, unităţi de execuţia branch-urilor, etc) conectate printr-o magistrală, plus regiştrii şi memorii cache. În cazul procesoarelor VLIW compilatorul are un rol decisiv în gruparea mai multor instrucţiuni independente într-o singură instrucţiune multiplă astfel încât să poată fi folosite eficient toate unităţile de execuţie pe perioada fiecărui impuls de tact.

[17] EPIC – calculatoare care exploatează paralelismul în mod explicit. Procesorul nu trebuie să asigure paralelismul instrucţiunilor. Compilatorul este responsabil pentru determinarea instrucţiunilor care pot fi executate în paralel şi codifică aceste informaţii direct în codul obiect obţinut după compilare).

Sursele FORTRAN ale programelor de test proprii (*benchmark-uri*) sunt convertite în C folosind translatorul *f2c*. Atât benchmark-urile C cât şi cele convertite din FORTRAN sunt compilate cu ajutorul compilatorului GNU *gcc* (dedicat arhitecturii SimpleScalar) rezultând cod în format de asamblare pentru respectiva arhitectură. Arhitectura SimpleScalar este derivată din arhitectura procesorului MIPS-IV ISA. Organizarea memoriei în sistemele bazate pe arhitectura SimpleScalar este convenţională. Spaţiul de adrese utilizator (pe 31 de biţi) este compus din trei părţi: cod, date şi stivă program. Instrucţiunile sunt pe 64 de biţi iar setul de regiştrii generali pentru numere întregi conţine 32 de regiştrii pe 32 de biţi fiecare.

Codul (în format asamblare) rezultat este trecut prin asamblorul SimpleScalar *as* care generează un format foarte asemănător cu codul obiect dar nu identic (cuprinde simboluri + cod obiect; nu este generat codul obiect pentru funcţiile de bibliotecă şi directivele de asamblare). Formatul final de cod obiect se obţine cu ajutorul utilitarului SimpleScalar *ld* care link-editează codul rezultat la pasul anterior (*pseudo-obiect*) cu bibliotecile proprii arhitecturii SimpleScalar (SS *libc.a*, SS *libm.a*, SS *libF77.a*). Codul obiect se constituie ca parametru de intrare pentru simulatoarele arhitecturii SimpleScalar. Setul standardizat de instrumente destinat cercetătorilor în microarhitecturi – SimpleScalar (o colecţie de compilatoare, asambloare, link-editoare, simulatoare), disponibil gratuit [Flo05], pune la dispoziţie şi câteva din benchmark-urile SPEC '95 în acelaşi format cod obiect (precompilate pentru arhitectura SimpleScalar).

Compilarea şi Link-editarea programelor

La ora actuală există pe piaţă o varietate de compilatoare disponibile, atât pentru platformă Windows cât şi pentru platformă Linux. Cele mai utilizate sunt: *bcc* (*Borland C Compiler* – sub Windows), *gcc* (*GNU C Compiler* – sub Linux), *xgcc* (gcc recompilat pentru a genera cod maşină specific arhitecturii SimpleScalar). Pe lângă compilator, furnizorul acestuia oferă şi uneltele suplimentare necesare: preprocesoare, asambloare şi link-editoare. Compilatoarele oferă o mulţime de opţiuni de compilare, preprocesare, link-editare cu biblioteci (matematice, grafice, etc) şi inclusiv facilităţi de depanare (*gdb – GNU debugger*). Pentru compilarea programelor se recomandă specificarea opţiunilor **–Wall** pentru afişarea tuturor avertismentelor generate de compilator. Prin opţiunea **-c** este evitată link-editarea iar **`-g`** `invocă opţiunea de depanare, impunând compilatorului generarea unei tabele de simboluri.`

Ex: ***gcc –c –g –O*** **nume_fis1.c nume_fis2.c** => Sunt generate doar codurile pseudo-obiect ale tuturor

fişierelor sursă respective fără a efectua legăturile cu bibliotecile funcţiilor apelate.

Link-editarea se poate realiza ulterior cu comanda:

gcc **nume_fis1.o nume_fis2.o ..... *-o*** **nume_fis_executabil** => determină codul maşină (**.exe** pentru **Intel**, **.ss** pentru **SimpleScalar**) care poate fi executat pe arhitectura gazdă (pentru care s-a generat).

Se reaminteşte că opţiunea de compilare **–S** realizează preprocesarea şi compilarea codului sursă, iar opţiunea **–s** generează doar codul asamblare fără faza de preprocesare. De asemenea, în cazul în care fişierele sursă **c** şi **cpp** nu există în directorul unde se găseşte compilatorul **gcc, xgcc**, etc atunci se va tasta odată cu numele şi calea spre acest fişier.

./xgcc **nume_fisier.c *–S*** => fişier asamblare MIPS (s-a folosit xgcc)

./xgcc **nume_fisier_sursa.c *–o*** **nume_fisier_destinatie.ss** => fişier executabil MIPS

10.2. IMPLEMENTAREA GESTIUNII STIVELOR DE DATE ASOCIATE FUNCŢIILOR C

10.2.1. SUBPROGRAME. GENERALITĂŢI. FUNCŢII C

Prin definiţie un *subprogram* (ex. *funcţie* C, funcţie sau procedură în Pascal) reprezintă un ansamblu de date, variabile şi instrucţiuni scrise în vederea unor prelucrări (calcule, citiri, scrieri) şi care poate fi utilizat (rulat) doar dacă este apelat de un program sau de alt subprogram. Cu alte cuvinte, reprezintă un program mai mic, mai simplu decât cel principal, creat cu scopul apelării lui cel puţin o dată. Prin realizarea de subprograme se stabilesc premisele programării structurate conform ecuaţiei programării structurate enunţată de Niklaus Wirth: **„Structuri de date + Algoritmi = Program”**. Caracterul generic (se operează asupra unei varietăţi de date de

intrare) şi concis (subprograme, structurile repetitive şi cele de tip recursiv) de scriere a programelor conduce la unele avantaje:

- *Reutilizarea codului*: - o dată scris un subprogram poate fi utilizat de mai multe programe şi chiar de acelaşi program de mai multe ori.
- Noţiunea de subprogram stă la baza mecanismului de recursivitate şi a tehnicii de *divide et impera.*
- Facilitează elaborarea algoritmilor prin descompunerea problemei de rezolvat în altele mai simple, fluxul de execuţie al programului putând fi urmărit cu o mai mare uşurinţă.
- Permite dezvoltarea modulară a codului (separat şi independent) de către programatori independenţi.
- Reducerea numărului de erori care pot apare la scrierea programelor.

FUNCŢII C [Zah04, Neg97, Sto98]

Funcţia este un concept important în matematică şi în programare. În limbajul C prelucrările sunt organizate ca o ierarhie de apeluri de funcţii. Orice program trebuie să conţină cel puţin o funcţie, funcţia *main*. Funcţiile încapsulează prelucrări bine precizate şi pot fi reutilizate in mai multe programe.

În esenţă, o funcţie este alcătuită din:

- **Antet** – conţine numele funcţiei, lista parametrilor formali, tipul rezultatului.

Tip_returnat *nume_funcţie* (lista_parametrilor formali)

Identic ca şi în matematică o funcţie nu poate întoarce mai mult de un rezultat. Există situaţii (efectuarea unor acţiuni, etc) când o funcţie nu întoarce nimic (în aceste cazuri tip_returnat este *void*). Funcţiile care nu returnează nimic se numesc în alte limbaje de programare *proceduri*. Tipul returnat de funcţie poate fi orice tip de dată cu excepţia masivelor. Dacă se doreşte ca o funcţie să returneze un masiv atunci acesta trebuie înglobat în structuri de date declarate ca *struct*. Rezultatul funcţiei este returnat la întâlnirea instrucţiunii *return* din corpul funcţiei (*return* expresie). Trebuie ca tipul expresiei să coincidă cu tipul funcţiei. La întâlnirea instrucţiunii *return*, după atribuirea valorii, execuţia funcţiei se încheie şi se revine la funcţia care a apelat-o. În absenţa instrucţiunii *return*, execuţia funcţiei se încheie după execuţia ultimei instrucţiuni. În acest caz funcţia nu întoarce nici o valoare.

Cu toate că uneori poate fi vidă, de cele mai multe ori lista parametrilor formali (*Parameters* – în engleză) are următoarea formă:

Tip_1 *Parametru*$_1$, Tip_2 *Parametru*$_2$, ..., Tip_n *Parametru*$_n$

- **O instrucţiune compusă** – aceasta cuprinde declaraţiile variabilelor locale şi instrucţiunile propriu-zise.

Transmiterea parametrilor

Parametrii care se găsesc în antetul funcţiei se numesc formali (declaraţi în funcţia apelată), iar cei care se găsesc în instrucţiunea de apel se numesc efectivi (*Arguments* – în engleză) şi sunt declaraţi în (sub)programul apelant. Între parametrii formali şi cei efectivi trebuie să existe o anumită concordanţă, descrisă prin regulile următoare:

- Numărul parametrilor formali (şi ordinea acestora) trebuie să coincidă cu numărul (şi ordinea) parametrilor efectivi. Există şi excepţii însă: funcţii cu număr variabil de parametrii. În cazul acestora este obligatoriu ca cel puţin primul parametru să apară în lista parametrilor formali. Pentru parametrii variabili antetul funcţiei va conţine „**...**” după primul parametru formal.
- Tipul parametrilor formali trebuie să coincidă cu tipul parametrilor efectivi sau tipul parametrilor efectivi să poată fi convertit implicit către tipul parametrilor formali, la fel ca în cazul atribuirii.

Există două mecanisme de transmitere a parametrilor: prin **valoare** şi prin **referinţă**. Transmiterea parametrilor prin valoare se utilizează atunci când nu ne interesează ca, la întoarcerea din subprogram (funcţie), parametrul efectiv să reţină valoarea modificată acolo (în subprogram). În cazul transmiterii prin valoare parametrii efectivi trebuie să fie valori reţinute de variabile sau expresii. Transmiterea parametrilor prin referinţă se utilizează atunci când se doreşte ca, la întoarcerea din subprogram, variabila transmisă să reţină valoarea stabilită în timpul execuţiei subprogramului. În cazul transmiterii prin referinţă parametrii efectivi trebuie să fie referinţe la variabile. Dacă în cazul transmiterii parametrilor prin valoare în stivă este reţinută valoarea parametrilor efectivi, în cazul transmiterii prin referinţă, în stivă este salvată adresa variabilelor.

Funcţia *main*()

Un program C este compus dintr-o ierarhie de funcţii, orice program trebuind să conţină cel puţin funcţia *main*(), prima care se execută la lansarea programului C. Codul funcţiei se găseşte între acolade.

```
void main(void){
        /* programul principal aferent fiecărui program C */
}
```

Definirea şi declararea unei funcţii

Pentru a putea fi utilizată într-un program, o funcţie trebuie să fie definită (sau declarată) şi apelată. **A defini** un subprogram, înseamnă a scrie efectiv corpul funcţiei. Trebuie avut grijă de alegerea locului unde se defineşte subprogramul (dacă el nu a fost anterior declarat atunci definiţia trebuie să aibă loc înainte de apelul acestuia). **A declara** un subprogram înseamnă a-l anunţa. Un subprogram nedeclarat nu poate fi folosit. Definiţia unui subprogram ţine loc şi de declaraţie.

Definiţia unei funcţii are următoarea formă:

```
tip_rezultat_returnat  nume_funcţie  (lista_parametri_formali){//antetul
                                                              funcţiei
    definirea variabilelor locale
    prelucrări              // instrucţiuni
    întoarcere rezultat
}                           // intre { } este implementat corpul funcţiei
```

Funcţiile nu pot fi definite imbricat (ca în Pascal). Definiţia unei funcţii trebuie să corespundă declaraţiei funcţiei. Fiecare argument al funcţiei (matematice) capătă un nume, devine un parametru formal, chiar dacă în declaraţia funcţiei făcută înaintea (şi distinct de) definiţiei funcţiei numărul acestora a fost declarat variabil (prin construcţia sintactică „**…**”).

Dacă funcţia nu returnează nici un rezultat (se mai numeşte funcţie *void*) şi nu primeşte parametri, definitia va fi:

```
void nume_funcţie (void){//antetul funcţiei
    definirea variabilelor locale
    prelucrări              // instrucţiuni
}                           // intre { } este implementat corpul funcţiei
```

O funcţie void fără parametri poate prelucra variabilele globale şi cele locale. Dacă mai multe funcţii void fără parametrii trebuie să aiba acces la aceleaşi date, acestea trebuie să fie variabile globale.

Exemplul următor conţine definiţia funcţiei Factorial care calculează iterativ factorialul unui număr.

```
long Factorial(int n)
{
  int i;
  long result = 1;
  for (i = 1; i <= n; i++)
    result *= i;
  return result;
}
```

Declaraţia unei funcţii se face prin precizarea **prototipului funcţiei** (antetul acesteia): *tip_rezultat_returnat* nume_funcţie (*lista_parametri_formali*);

Prototipul (*declaraţia funcţiei*) implicit este:

int nume_functie(*void*);

Prototipul unei funcţii *f* trebuie să apară în afara tuturor funcţiilor şi înaintea oricărui apel al său. Prin intermediul prototipului compilatorul ştie cum trebuie apelată funcţia. Compilatorul află detalii referitoare la tipul returnat de funcţie, numărul şi tipul parametriilor de intrare (efectivi), numele funcţiei folosite ca şi parametru (în cazul apelurilor indirecte de funcţii prin pointer). Informaţiile sunt extrem de utile mai ales atunci când compilatorul trebuie să genereze cod pentru o funcţie care a fost apelată înaintea definirii sale în codul sursă. De asemenea, se poate întâmpla ca, în cazul proiectelor de aplicaţii care conţin mai multe fişiere sursă şi biblioteci (fişiere *header* şi sursă *C*), o funcţie să fie apelată într-un fişier sursă (scris de un programator) şi să fie definită într-altul (scrisă de alt programator). În aceste situaţii, fişierul în care este făcut apelul funcţiei trebuie să „*includă*” fişierul header cu declaraţia funcţiei şi este compilat separat de fişierul care conţine definiţia funcţiei, în final după faza de link-editare rezultând fişierul executabil ca un „tot unitar”.

Apelul unei funcţii:

nume_functie(lista_parametri_efectivi) /*poate apare ca operand într-o expresie, dacă funcţia returnează un rezultat*/

La apelul unei funcţii (subprogram), se realizează transferul controlului din (sub)programul apelant către subprogramul apelat şi se execută corpul acestuia, după care se revine în funcţia apelantă, la instructiunea următoare apelului. O funcţie poate fi apelată, dacă în faţa apelului există definiţia sau cel puţin declaraţia funcţiei. Pentru a realiza acest lucru, compilatorul

păstrează pe stiva de date aferentă funcţiei apelate adresa de revenire în (sub)programul apelant (vezi subcapitolul 10.2.2). În cazul funcţiilor *void* fără parametri, apelul se face prin:

nume_funcţie(); //instrucţiune care efectuează o acţiune

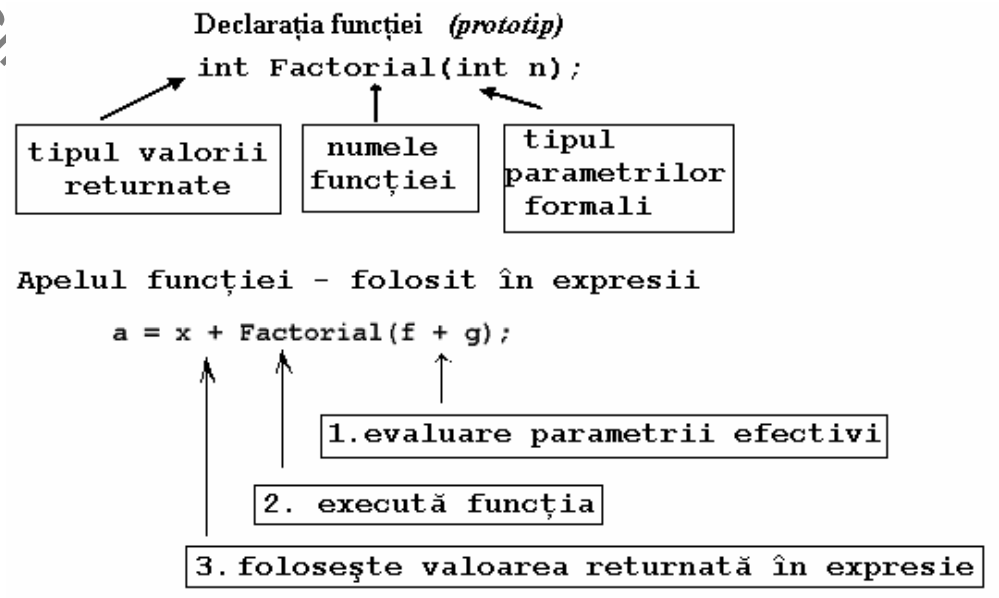


Figura 10.6. Exemplu de declaraţie şi apel a funcţiei *Factorial*

În continuare este prezentat un exemplu de program C care efectuează operaţii aritmetico-logice asupra unor variabile locale şi globale şi afişează pe ecran rezultatul. Acest program C este translatat în cod asamblare LC-3, arătându-se şi tabela de simboluri generată de compilator [Patt03].

Codul sursă *C*

```
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;   /* variabile locale funcţiei main */
  int outLocalA;
  int outLocalB;

  /* iniţializare */
  inLocal = 5;
  inGlobal = 3;
```

```
  /* efectuarea operaţiilor */
  outLocalA = inLocal++ & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* afişare rezultate */
  printf("The results are: outLocalA = %d outLocalB = %d \n",
outLocalA, outLocalB);
}
```

Tabela de simboluri

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	3	main
outLocalA	int	4	main
outLocalB	int	5	main

Variabilele globale startează la offset 0
Variabilele locale startează la offset 3

Generarea codului asamblare (C -> LC-3)
; funcţia *main* şi iniţializarea variabilelor

```
    AND R0, R0, #0
    ADD R0, R0, #5  ; inLocal = 5
    STR R0, R6, #3  ; (offset = 3)

    AND R0, R0, #0
    ADD R0, R0, #3  ; inGlobal = 3
    STR R0, R5, #0  ; (offset = 0)
```

; calculul primei expresii

```
; outLocalA = inLocal++ & ~inGlobal;
    LDR R0, R6, #3  ; get inLocal
    ADD R1, R0, #1  ; increment
    STR R1, R6, #3  ; store

    LDR R1, R5, #0  ; get inGlobal
    NOT R1, R1      ; ~inGlobal
    AND R2, R0, R1  ; inLocal & ~inGlobal
    STR R2, R6, #4  ; store in outLocalA                ; (offset = 4)
```

```
; calculul celei de-a doua expresii
; outLocalB = (inLocal + inGlobal)        ;- (inLocal - inGlobal);
    LDR R0, R6, #3                       ; inLocal
    LDR R1, R5, #0                       ; inGlobal
    ADD R0, R0, R1                       ; R0 este suma
    LDR R2, R6, #3                       ; inLocal
    LDR R3, R5, #0                       ; inGlobal
    NOT R3, R3
    ADD R3, R3, #1
    ADD R2, R2, R3                       ; R2 este diferența (calculată
                                         ca o adunare în complement
                                         față de 2)
    NOT R2, R2                           ; negație
    ADD R2, R2, #1
    ADD R0, R0, R2                       ; R0 = R0 - R2
    STR R0, R6, #5                       ; outLocalB (offset = 5)
```

10.2.2. STIVA DE DATE AFERENTĂ FUNCȚIILOR [Patt03, Vin03]

După cum se ştie, stiva de date asociată unei funcţii scrise în limbajul C reprezintă o zonă de memorie unde sunt stocate toate variabilele locale şi parametrii aferenți respectivei funcţii (”*activation record*”). Fiecare funcţie C are propriul context păstrat în această structură de date specială. Stiva de date se asociază în mod dinamic fiecărei funcţii în curs. Aşadar, o funcţie poate avea la un moment dat mai multe stive (instanţe), doar una fiind însă activă. Spre exemplu, recursivitatea se poate implementa facil în C tocmai datorită acestei caracteristici.

Structura stivei de date asociate unei funcţii C este prezentată în figura următoare, corespunzător secvenței de program de mai jos.

```
int NoName (int a, int b){
    int w,x,z;
    /* Corpul funcției */
            .
            .
    return y;
}
```

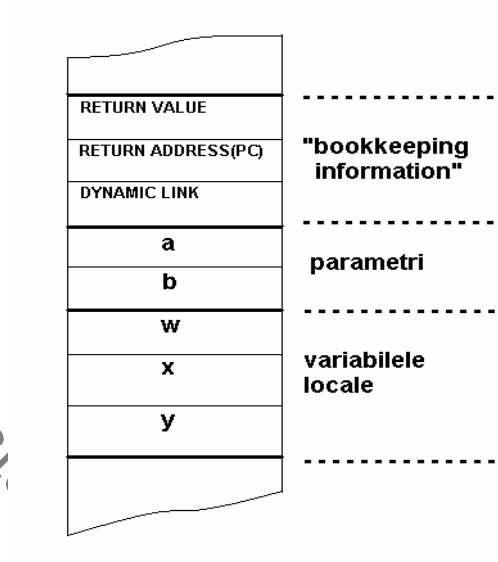


Figura. 10.7. Structura stivei de date asociate funcţiei NoName

RETURN VALUE: aici se plasează valoarea variabilei y, chiar înainte de revenirea din funcţie. Acest câmp există şi dacă funcţia nu ar returna practic nici o valoare.

RETURN ADDRESS (PC): reprezintă PC-ul de revenire în funcţia apelantă.

DYNAMIC LINK: memorează adresa de început a stivei de date aferente funcţiei apelante. În continuare, se va considera că registrul R6 va conţine adresa de început a stivei de date asociată funcţiei respective.

Aşadar funcţia *NoName* procesează asupra parametrilor trimişi de către funcţia apelantă (a,b) respectiv asupra variabilelor locale ale acesteia (w,x,y). Firește, ea trimite rezultatele (y) către funcţia apelantă prin stiva de date curentă. În continuare se consideră că atunci când funcţia *NoName* este apelată, pointerul la stivele de date (R6, aici) va pointa la începutul stivei de date aferente funcţiei.

Pentru a înţelege implementarea apelurilor/revenirilor funcţiilor, se consideră următorul exemplu de program:

```
main()
{
 int a;
 int b;
 .
```

```
.
b=NoName(a,10);
.
.
}
int NoName(int a, int b)
{
  int w,x,y;
        /* Corpul funcţiei */
        .
        .
        return y;
}
```

Stiva de date începe la o locaţie de memorie determinată de către proiectanţii sistemului de operare şi creşte înspre adrese crescătoare. Execuţia programului începe cu un apel al sistemului de operare către funcţia "main". În acest punct, stiva datelor funcţiei "main" se structurează în memorie, iar registrul R6 pointează la începutul ei. În translatarea apelului unei funcţii, compilatorul generează automat cod maşină pentru a înscrie o stivă de date în memorie. În translatarea revenirii dintr-o funcţie apelată în funcţia apelantă, compilatorul generează automat cod pentru preluarea stivei de date din memorie.

Apelul şi revenirea se fac în 4 paşi, aşa cum se prezintă în continuare :

1) Apelul funcţiei NoName

Se face prin asignarea b=NoName(a,10); Stiva de date a funcţiei main() respectiv a funcţiei NoName(a,10) sunt prezentate în figura următoare.

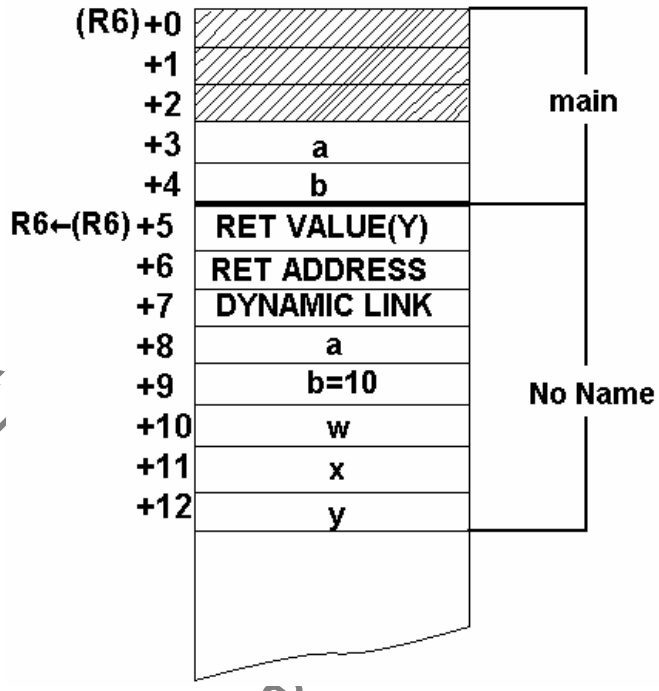


Figura 10.8. Stivele de date asociate funcţiilor main() şi NoName(a,10)

Secvenţa compilată a apelului funcţiei NoName este următoarea:

```
        ld R8, (R6)3;       R8 ← a
        st R8, (R6)8;       a → Stiva NoName
        and R8, (R8), #0    ;R8 ← 0
        add R8, R8, #10;    R8 ← 10
        st R8, (R6)9;       (b=10) → Stiva NoName
        st R6, (R6)7;       (R6)main → Stiva NoName (Dynamic
                            Link)
        add R6, R6, #5;     R6 ← (R6) +5, actualizare nou început
                            al stivei de date
        jsr NoName;         apel funcţie, R7←PCnext şi
                            PC←(NoName)
PCnext: ld R8, (R6)5
        st R8, (R6)4
```

2) <u>Startarea funcţiei apelate (NoName)</u>

Începe cu instrucţiunea care salvează în stiva de date a funcţiei *NoName* adresa de revenire în funcţia principală main(). Adresa de revenire

se află stocată în registrul R7 (conține adresa următoare a instrucțiunii JSR NoName)

```
st R7, (R6)1
```

3) Sfârşitul funcţiei apelate (NoName)

```
ld R8, (R6)7
st R8, (R6)0;     Se memorează valoarea lui y în RET VALUE
                  din stiva de date
ld R7, (R6)1;     R7← RET ADDRESS
ld R6, (R6)2;     R6 ← adresa de început a stivei de date
                  aferente funcției main ().
RET;              PC← adresa de revenire în funcția main().
```

4) Revenirea în funcţia apelantă (main)

```
          JSR NoName;  pe această instrucțiune se face revenirea (la
                       finele execuției acesteia)
PCnext:   ld R8, (R6)5;  R8←valoarea lui y din funcția NoName
          st R8, (R6)4;  se face asignarea : b=NoName(a,10).
```

Obs.: Stivele de date asociate funcțiilor C sunt structuri de tip tablou, având o adresă de bază stocată în registrul R6 şi un număr (variabil) de elemente. Având în vedere frecvența deosebită a accesării acestor structuri de date, modul de adresare indexat ($R_{bază}$ + offset) este esențial în facilitarea manipulării datelor prin aceste structuri de tip tablou [Vin06].

10.3. EXERCIŢII ŞI PROBLEME

1. Completați cu instrucțiunile C corespunzătoare în tabelele din partea dreaptă astfel încât din punct de vedere logic să se realizeze acelaşi lucru ca şi în stânga (cel descris de secvența asamblare LC-3).

a)

```
                                main()
                                {
                                      int   v1;
                                      int   v2;
                                      int   temp;
LDR R0, R6, #3
LDR R1, R6, #4
STR R0, R6, #4
STR R1, R6, #3
                                }
```

b)

```
                                main()
                                {
                                      int   v3;
                                      int   v4;
                                      int   v5;
LDR R0, R6, #3
NOT R0, R0
LDR R1, R6, #4
NOT R1, R1
AND R2, R0, R1
NOT R2, R2
STR R2, R6, #5
                                }
```

c)

```
                                main()
                                {
        AND R0, R0, #0                int   i;
        ADD R0, R0, #15               int   j;
        STR R0, R6, #4                int   k;
FL1     AND R0, R0, #0
        STR R0, R6, #3
FL1_2   LDR R0, R6, #3
```

LDR R1, R6, #4
NOT R1, R1
ADD R1, R1, #1
ADD R0, R0, R1
BRzp DONE
F LDR R0, R6, #5
ADD R0, R0, #1
STR R0, R6, #5
 	
LDR R0, R6, #3
ADD R0, R0, #2
STR R0, R6, #3
BRnzp FL1_2
DONE …
…
	}

2. Referitor la sintaxa limbajului *C* răspundeți la următoarele întrebări legate de funcții:
a) Ce reprezintă declarația unei funcții? Care este scopul declarării?
b) Ce este prototipul unei funcții?
c) Ce reprezintă definiția unei funcții?
d) Ce sunt parametrii efectivi ai unei funcții? Dar parametrii formali ai funcției?

3. Ce valori sunt afișate pe ecran în urma execuției programului de mai jos? Ce modificări trebuie aduse codului sursă pentru a afișa pe ecran valorile: 2 3?

```
#include <stdio.h>
void MyFunc(int z);
main()
{
 int z=2;
 MyFunc(z);
 MyFunc(z);
}

void MyFunc(int z)
{
```

```
 printf("%d ",z);
 z++;
}
```

4. Cele două secţiuni de cod de mai jos (una în C, cealaltă în asamblare LC-2) trebuie să realizeze acelaşi lucru. Completaţi cele trei spaţii goale cu câte o instrucţiune fiecare.

int x; int *y; int z;	
x = 0;	LD R0, X AND R0, R0, #0 ST R0, X
. .	. .

x++;	LD R0, X
	ADD R0, R0, #1
	_____

_____	LEA R0, X ST R0, Y
z = (*y) + 1;	_____ ADD R0, R0, #1 ST R0, Z

5. Se consideră următorul program care citeşte de la tastatură două caractere şi afişează maximul dintre ele prin intermediul funcţiei Max.

```
#include <stdio.h>
char Max ( char, char ); /* prototype */
int main ( ) /* main returns an int */
{
     char ch1,ch2,answer;

     printf ("Welcome to Max of 2 Chars!\n");
     printf ( "Enter 2 characters:\n" );
     scanf ( "%c%c", &ch1, &ch2);
     answer = Max(ch1, ch2);
     printf ("Max is : %c", answer);
     return 0;
} /* end of main function */
```

```
char Max (char ch1, char ch2)
{
     char higher;
     if ( ch1 > ch2 )
       higher = ch1;
     else
       higher = ch2;
     return higher;
}
```

a) Realizaţi tabela de simboluri generată de compilator pentru programul dat.
b) Presupunând că de la tastatură se introduc caracterele ‘A’ şi ‘Q’ şi că valoarea iniţială a registrului R6 (*stack pointer*) este 0x4000, descrieţi stiva de date aferentă funcţiilor *main*() şi *Max*() în câteva momente importante: (iniţial când doar funcţia *main* este activă, după citirea celor două caractere de la tastatură când şi funcţia *Max* devine activă – la apel, când funcţia *Max* este activă înainte de revenire când există rezultatul disponibil, după predarea rezultatului de către *Max* când doar funcţia *main* a mai rămas activă, înainte de terminarea programului).
c) Realizaţi translatarea în cod asamblare LC-3 a funcţiei *Max* din *C*.

11. INTRODUCERE ÎN *RECURSIVITATE*. COMPARAȚIA DINTRE *RECURSIV* ŞI *ITERATIV* ÎN ALEGEREA ALGORITMULUI DE REZOLVARE A PROBLEMELOR. AVANTAJE / DEZAVANTAJE

11.1. SCOP ŞI COMPETENȚE NECESARE

Capitolul de față își propune o introducere în mecanismul recursivității, bazându-se pe noțiunile anterioare de funcție (subprogram, subrutină, etc) şi stivă de date aferentă funcției. Competențele generale care trebuiesc dovedite de către cei care citesc acest curs se referă la capacitatea de a descompune o problemă în subprobleme, de prelucrare a datelor utilizând subprograme şi respectiv de a aplica anumiți algoritmi în prelucrarea structurilor de date. Scopul lucrării este de cunoaştere şi înțelegere a mecanismului recursivității (după unii autori recursivitatea pare „*magică*", dincolo de înțelegere). Se va analiza comparativ modul de rezolvare recursiv şi iterativ al unor probleme. Se va lămuri folosind diferite exemple necesitatea aplicării condiției de consistență şi în ce situații este utilă rezolvarea recursivă a unor probleme.

11.2. RECURSIVITATEA

Reprezintă o **noțiune fundamentală a informaticii** şi face parte din domeniul general al algoritmilor şi structurilor de date. Constituie o tehnică de programare care permite o **exprimare extrem de concisă şi clară** a algoritmilor de rezolvare a unor probleme complexe. Recursivitatea din programare este derivată în mod natural (din necesități practice) din noțiunea matematică [Cor90, Sto98]. Astfel, **în matematică o noțiune este**

definită recurent (recursiv) dacă în cadrul definiției apare însăşi noțiunea care se defineşte. La scrierea unui algoritm recursiv este suficient să gândim ce se întâmplă la un anumit nivel, pentru că la orice nivel se întâmplă exact acelaşi lucru. Recursivitatea în programare – a apărut după anii ’80 odată cu limbajele de nivel înalt moderne Algol, Pascal, C. Fortran şi Cobol **nu** permiteau scrierea programelor recursive. Un motiv poate fi de natură istorică. **Capacitatea redusă a memoriei sistemelor din anii 70 – 80** a condus la constrângeri privind execuția programelor procedurale într-un spațiu mic de adrese (de memorie) şi alocări reduse pe stivă sau în “heap”. Recursivitatea reprezintă un mecanism general de elaborare a programelor constând în **posibilitatea ca un subprogram să se autoapeleze**. Există două tipuri de recursivitate:

- **directă**: dacă apelul subprogramului apare chiar în corpul său. Ex: *factorial, fibonacci*, *fractali* precum şi alte probleme din medicină – informația genetică conținută în nucleul unei celule se repetă la diferite scări.
- **indirectă**: dacă apelul subprogramului recursiv apare în instrucțiunea (compusă) a unui alt subprogram care se apelează direct sau indirect din subprogramul recursiv. Ex: *evaluarea unei expresii, forma poloneză prefixată a unei expresii*.

11.2.1. STIVA DE DATE ASOCIATĂ UNEI FUNCȚII [Patt03]

- Stă la baza recursivității.
- Zonă a memorie de date care funcționează după principiul LIFO.
- Este gestionată implicit de către compilator.
  - Stiva de date începe la o locație de memorie determinată de către proiectanții sistemului de operare şi creşte înspre adrese descrescătoare. În limbajul C, execuția programului începe cu un apel al sistemului de operare către funcția “main”. În acest punct, stiva datelor funcției “main” se structurează în memorie, iar un registru pointează la începutul ei.
  - În translatarea **apelului unei funcții**, compilatorul generează automat cod maşină pentru **a înscrie o stivă de date în memorie**. În translatarea r**evenirii dintr-o funcție apelată în funcția apelantă**, compilatorul generează automat cod pentru **preluarea stivei de date din memorie**.

- Pentru fiecare (auto)apel se creează un alt nivel pe stiva de date, în care se depun noile valori – parametrii noului apel.

- La fiecare apel de funcție (procedură) sunt salvate automat în stivă (nu neapărat în această ordine, în funcție de ISA-ul fiecărui procesor):
  - Valorile parametrilor de tip valoare;
  - Adresele parametrilor de tip referință;
  - Variabilele locale ale subprogramului;
  - Adresa de revenire în (sub)programul apelant;

Exemplu:

Structura stivei de date asociate unei funcții C este prezentată în figura următoare, corespunzător secvenței de program de mai jos.

```
int NoName (int a, int b){
    int w,x,y;
    /* Corpul funcției */
    .
    return w;
}
```

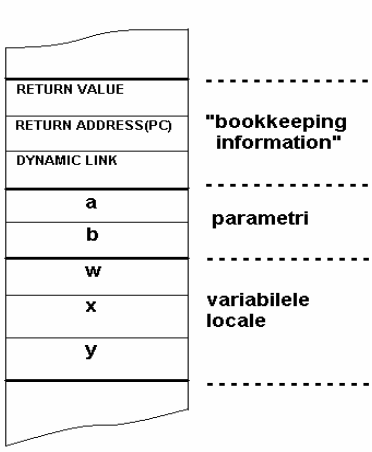


Figura 11.1. Structura stivei de date asociate funcției *NoName* în cadrul arhitecturii LC-3

RETURN VALUE: aici se plasează valoarea variabilei *w*, chiar înainte de revenirea din funcție. Acest câmp există şi dacă funcția nu ar returna practic nici o valoare.

RETURN ADDRESS (PC): reprezintă PC-ul de revenire în funcția apelantă.

DYNAMIC LINK: memorează adresa de început a stivei de date aferente funcției apelante.

- Orice subprogram recursiv trebuie să satisfacă o **condiție de consistență** (procesul de autoapelare trebuie să se oprească după un număr finit de paşi. Altfel va apărea eroarea „***Stack Overflow***!”).

- Exemplu de funcţie **inconsistentă**:

$$incons(n) = \begin{cases} 1, & \text{dacă } n = 0 \\ n \cdot incons(n+1), & \text{dacă } n > 0 \end{cases}$$

întrucât nu pot fi calculate valorile funcţiei pentru n>0.

Pentru a înţelege implementarea apelurilor / revenirilor funcţiilor, se consideră cel mai simplu şi reprezentativ exemplu de subprogram recursiv factorialul unui număr natural *n*.

$$fact(n)=n!=n\cdot(n-1)!$$

Implementare **iterativă**

```
long factorial(int n){
  int i;
  long f=1;

  for(i=1;i<=n;i++)
          f=f*i;
  return f;
}
```

Implementare **recursivă**

```
long fact(int n){
  if(n<=1)
          return 1;
  else
          return n*fact(n-1);
}
```

Return Value	Main
Return Address	
Dynamic Link	
Return Value (***fact(n)***)	fact(4)
Return Address	
Dynamic Link	
n (4)	
Return Value (***fact(n-1)***)	fact(3)
Return Address	
Dynamic Link	
n-1 (3)	
Return Value (***fact(n-2)***)	fact(2)
Return Address	
Dynamic Link	
n-2 (2)	
Return Value (***fact*(1) = 1**)	fact(1)
Return Address	
Dynamic Link	
n-3 (1)	

Creşterea stivei

Figura 11.2. Imaginea stivei pentru calculul valorii fact(4) în implementarea recursivă

În continuare sunt ilustrate două exemple în care se impune rezolvarea recursivă a problemei (algoritmul de *căutare binară* şi respectiv problema *turnurilor din Hanoi*). Avantajul, în cazul primului exemplu îl reprezintă şi timpul logaritmic de execuție față de cel polinomial (liniar) dacă rezolvarea problemei ar fi fost iterativă (căutare directă – parcurgere tablou şi comparare cu fiecare element).

Căutare Binară

Considerându-se o grupă de studenți ordonată alfabetic se cere ca printr-un număr cât mai mic de căutări să se determine dacă un anumit student se află în listă sau nu. Algoritmul de rezolvare presupune următorii paşi:

0. Dacă lista are cel puțin 2 studenți atunci se execută paşii, altfel înseamnă că nu s-a găsit studentul căutat în listă:

1. Se verifică dacă studentul căutat este cel din mijlocul listei.

2. Dacă rezultatul căutării este cu succes algoritmul se încheie, dacă nu atunci urmează:

3a. Dacă numele studentului căutat este mai mare (alfabetic) decât al studentului din mijlocul listei atunci algoritmul de căutare se reaplică pe jumătatea superioară a listei (de la mijloc spre sfârşitul alfabetului).

3b. Dacă numele studentului căutat este mai mic (alfabetic) decât al studentului din mijlocul listei atunci algoritmul de căutare se reaplică pe jumătatea inferioară a listei (de la începutul alfabetului spre mijloc).

Căutare Binară - Pseudocod

```
FindExam(studentName, start, end){
    halfwayPoint = (end + start)/2;
    if (end < start)
        ExamNotFound();                 /* nu mai am decât cel mult un
                                           student în listă */
    else if (studentName == NameOfExam(halfwayPoint))
        ExamFound(halfwayPoint);        /* Student găsit! */
         else if (studentName < NameOfExam(halfwayPoint))
                                        /*caută în jumătatea
                                           inferioară*/
                FindExam(studentName, start, halfwayPoint - 1);
              else /* caută în jumătatea superioară */
                FindExam(studentName, halfwayPoint + 1, end);
}
```

Turnurile din Hanoi

Se dau trei tije simbolizate prin A (sursă), C (destinaţie) şi B (manevră). Pe tija A se găsesc n discuri de diametre diferite, aşezate în ordine descrescătoare a diametrelor privite de jos în sus (discul 1 se află în vârf iar n este la bază). Se cere să se mute discurile de pe tija A pe tija C, folosind tija B ca tijă de manevră, respectându-se următoarele reguli:

- La fiecare pas se mută un singur disc.
- Nu este permis să se aşeze un disc cu diametrul mai mare peste un disc cu diametrul mai mic.

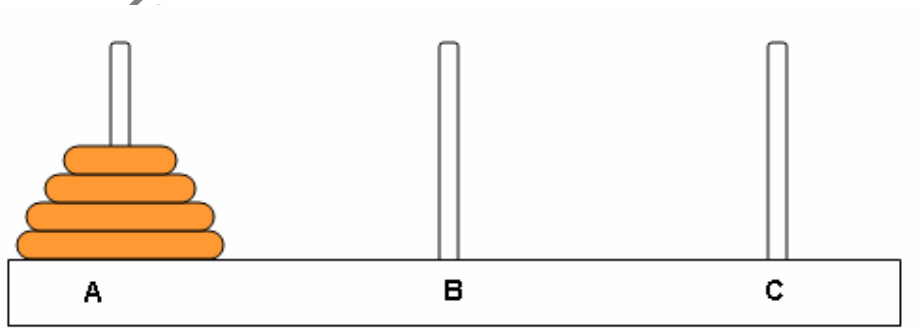


Figura 11.3. Turnurile din Hanoi – Configuraţia iniţială

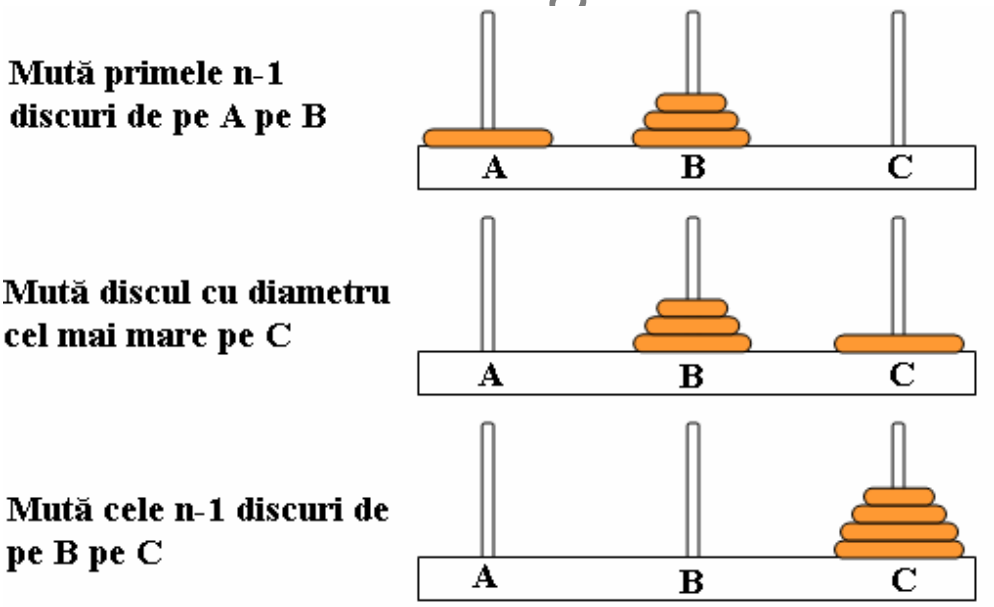


Figura 11.4. Turnurile din Hanoi – Descompunerea problemei în subprobleme

De fapt, prima subproblemă este identică cu problema inițială, cu deosebirea că sunt mai puține discuri de mutat şi tijele şi-au schimbat ordinea (A este sursă, B este destinație şi C manevră).

- "Mută primele n-1 discuri de pe A pe B."

A doua subproblemă, şi ea coincide în mare parte cu problema inițială, deosebirea reprezentând-o numărul mai mic de discuri de mutat şi ordinea tijelor (B este sursă, C este destinație şi A manevră).

- " Mută cele n-1 discuri de pe B pe C."

Se observă astfel caracterul recursiv al problemei. Condiția de ieşire din recursivitate este atunci când n=1 (un singur disc care se va muta de pe sursă pe destinație). Pentru această operație nu este nevoie de tijă de manevră.

- " Mută discul cu diametrul cel mai mare de pe A pe C."

```
MoveDisk(diskNumber, startPost, endPost, midPost){
        if (diskNumber > 1) {
            /* Mută primele n-1 discuri de pe startPost pe midPost */
            MoveDisk(diskNumber-1, startPost, midPost, endPost);
            printf("Muta discul %d de pe %d pe %d.\n", diskNumber,
        startPost, endPost);
            /* Mută cele n-1 discuri de pe midPost pe endPost.*/
            MoveDisk(diskNumber-1, midPost, endPost, startPost);
        }
        else
            printf("Muta discul 1 de pe %d pe %d.\n", startPost, endPost);
}
```

Observații:

- **Pentru orice algoritm *recursiv* există unul *iterativ* care rezolvă aceeaşi problemă**.
- Rezolvarea **recursivă** a problemelor **simplifică munca programatorului dar complică lucrurile la nivelul codului obiect deoarece operațiile cu stiva presupun un consum suplimentar de timp şi memorie, timpul necesar de calcul pentru algoritmul iterativ fiind mult mai mic**.
- **Consecință**: **Nu întotdeauna alegerea unui algoritm recursiv reprezintă un avantaj**.

- Exemplificare: Şirul lui Fibonacci – şir recurent de ordinul 2.

$$fib(n)=\begin{cases} 1, & \text{dacă } n=0 \text{ sau } n=1 \\ fib(n-2)+fib(n-1), & \text{dacă } n>1 \end{cases}$$

O implementare C **recursivă** a calculului elementului *fib(n)* din cadrul acestui şir recurent este prezentată mai jos (stânga):

```
#include<stdio.h>
int fib(int n);
main (){
    int in;
    int numar;
    printf("Care termen din şir?");
    scanf("%d", &in);
    numar = fib(in);
    printf("Termenul are valoarea
            %d\n", numar);
}

int fib(int n)
{
    if (n = = 0 || n = = 1)
        return 1;
    else
        return (fib(n-1) + fib(n-2));
}
```

Implementare **iterativă**

```
#include<iostream.h>
main (){
        int n, f0=0,f1=1,f2;
        cout<<"n=";cin>>n;
        if(!n)
            cout<<f0;
        else
            if(n==1)
                cout<<f1;
            else{
                for(i=2;i<=n;i++){
                    f2=f0+f1;
                    f0=f1;
                    f1=f2;
                }
            cout<<f2;
            }
}
```

- **Avantaje:**
  - Şirul Fibonacci este utilizat în probleme de sortare, căutare şi strategii de joc (IA).
  - A fost folosit în matematică de Edmond Lucas pentru a arăta că $2^{127}-1$ este număr prim [Cor90].

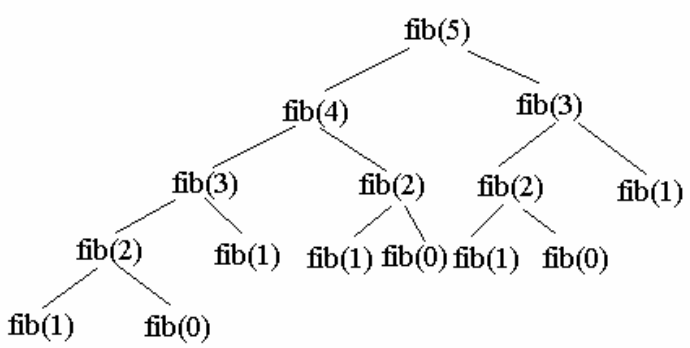


Figura 11.5. Ineficiența recursivității în cascadă – calculul unor termeni de foarte multe ori

- **Dezavantaje (*recursivitate în cascadă*):**
  - Pentru calculul lui *fib*(*n*) este necesar să se cunoască *fib*(*n-1*) şi *fib*(*n-2*). Parametrii acestor funcții sunt depuşi în stivă. Procedeul continuă până este calculat *fib*(*n-1*), apoi se reia calculul lui *fib*(*n-2*). Acest lucru este extrem de ineficient pentru valori mari ale lui *n* (*n>100*). În figura următoare 11.6 se poate observa *trace-ul apelurilor funcției fib*(*3*) – modul de transmitere al argumentelor şi ordinea de calcul a termenilor**.**
  - Varianta de calcul recursiv presupune recalcularea unei valori de mai multe ori, **varianta iterativă fiind liniară**. Astfel, pentru calculul lui *fib*(5) termenul *fib*(3) a fost calculat de 2 ori iar termenul *fib*(2) a fost calculat de 3 ori determinând **creşterea stivei** prin faptul că există instanțe dinamice multiple ale funcției apelate. Dezavantajul este mult mai acutizat pentru valori foarte mari ale lui *n* – „*De câte ori este calculat şi folosit fib*(2) *în calculul lui fib*(100) ?” În aceste situații (recursivitate în cascadă), o soluție posibilă (neimplementată încă hardware în microprocesoarele comerciale) o reprezintă reutilizarea dinamică a instrucțiunilor (rezultatul acestora) atât la nivel de funcție – *coarse grain* cât şi la nivel de instrucțiune mașină – *fine grain*.

fib(3)
fib(2)
fib(1)
fib(0)
fib(1)

Figura 11.6. Trace-ul apelurilor funcției fib(3)

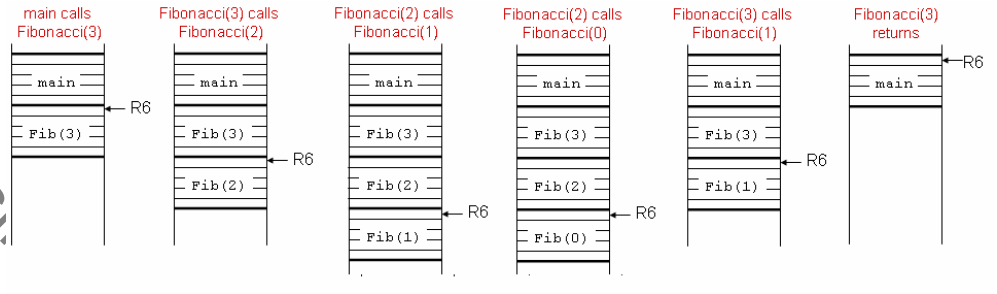


Figura 11.7. Variaţia stivei (şi a indicatorului de stivă – R6) în cazul apelurilor funcţiei fib(3)

11.3. IMPLEMENTAREA RECURSIVITĂŢII LA NIVELUL STIVEI DE DATE

Esenţa implementării recursivităţii constă în manipularea stivelor de date asociate funcţiilor dinamice (adică funcţiilor în curs de execuţie la un moment dat). La implementarea algoritmilor este uzual să asociem obiecte locale funcţiilor, precum variabile, constante, tipuri, obiecte care nu au semnificaţie în afara funcţiei respective. **Definiţiile parametrilor funcţiilor recursive, a variabilelor locale şi globale sunt fundamentale pentru execuţia programului** [Patt03]. Astfel, variabilele folosite pentru a parcurge valorile permise componentelor unei soluţii trebuie să fie locale funcţiei, altfel nu se generează corect soluţia. Definiţiile bine alese pot optimiza spaţiul de memorie alocat pe stivă şi timpul de execuţie. De fiecare dată când o funcţie este apelată recursiv, pe stiva sistemului se crează un nou set de variabile locale. Deşi noile variabile locale au acelaşi nume cu cele existente înainte de activarea funcţiei, valorile lor sunt distincte, şi orice conflict de nume este evitat de aplicarea regulilor domeniilor de vizibilitate a identificatorilor. Aceste reguli spun că identificatorii se referă întotdeauna la ultimul set de variabile creat.

În continuare se prezintă, la nivel de cod obiect, implementarea funcţiei Fib(int n), cu referire la stivele de date accesate [Vin03, Patt03].

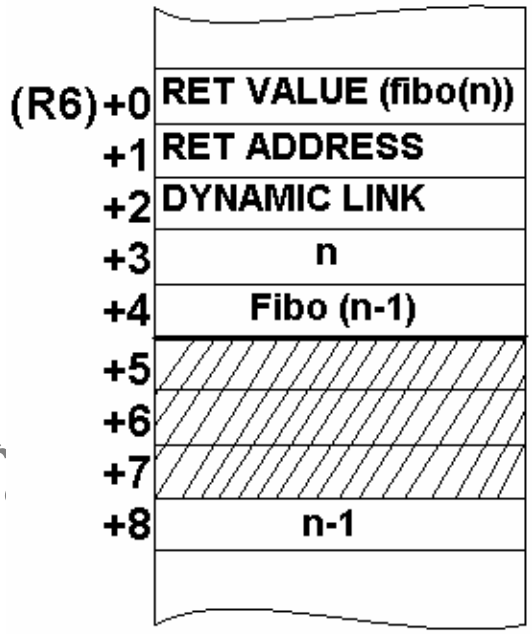


Figura 11.8. Stiva de date asociată funcției Fib(n)

Fib:

```
STR R7, R6, #1;        salvează PC revenire în stiva de date
LDR R0, R6, #3;        R0 ← valoare "n"
BRz Fib_end;           dacă n=0 atunci se încheie recursivitatea,
                       fib(0)=1
ADD R0,R0, # -1
BRz Fib_end;           şi dacă n=1 se încheie recursivitatea, fib(1)=1

; temp = fib(n-1)
LDR R0,R6, #3;         se calculează n-1 R0 ←n
ADD R0,R0, # -1;       R0 ← n-1
STR R0, R6, #8;        pune (n-1) ca parametru în stiva funcției
                       fib(n-1)
STR R6, R6, #7;        pune adresa de început a stivei funcției fib(n)
                       în stiva funcției fib(n-1)
ADD R6, R6,#5;         pune în R6 noua adresă de început aferentă
                       stivei de date a lui fib(n-1)
JSR Fib;               apel funcție fib (recursiv)
LDR R0, R6, #5;        R0 ← valoarea returnată de fib(n-1)
```

```
        STR R0, R6, #4;     memorează variabila locală aferentă stivei de
                            date a lui fib(n)

        ; R0 = fib(n-2)
        LDR R0, R6, #3;     R0 ← (n-1)
        ADD R0, R0, #-1;    R0 ← (n-2)
        STR R0, R6, #8
        STR R6, R6, #7
        ADD R6, R6, #5;     pregăteşte stiva de date a lui fib(n-2)
        JSR Fib;            apel recursiv
        LDR R0, R6, #5;     R0 ← fib(n-2)

        ; return R0 + temp
        LDR R1, R6, #4;     R1 ← fib(n-1)
        ADD R0, R0, R1;     R0 ← fib(n-1) + fib(n-2)
        STR R0, R6, #0;     fib(n) → RET VALUE
        LDR R7, R6, #1;     reface R7 (adresa de revenire în programul
                            apelant)
        LDR R6, R6, #2;     revenire în stiva precedentă
        RET
Fib_end:
        AND R0, R0, #0      ; set R0 = 1
        ADD R0, R0, #1
        STR R0, R6, #0      ; store 1 to return value
        LDR R7, R6, #1;      reface R7, R6 (return address şi stack
                            pointer)
        LDR R6, R6, #2
        RET
```

Obs: Legătura între arhitectura unui microprocesor şi aplicaţiile scrise în limbaje de nivel înalt este una complexă şi extrem de subtilă. Există microarhitecturi de calcul optimizate în mod special în vederea rulării eficiente a unor clase de aplicaţii bine precizate scrise în anumite limbaje de nivel înalt (ex. microprocesoare Java) [Patt03].

- **Concluzie: Este necesară o comparaţie între cele două moduri (*iterativ* şi *recursiv*) de rezolvare a problemei date, nu doar văzut avantajul introdus de simplitatea şi compactitatea codului din punct de vedere al programatorului.**

11.4. TIPURI DE FUNCȚII RECURSIVE. ELIMINAREA RECURSIVITĂȚII

Recursivitate liniară

Se caracterizează prin faptul că două apeluri recursive ale lui ***f*** pot apare numai în ramificații diferite ale aceleiaşi alternative.

Recursivitatea liniar repetitivă

Este un exemplu de recursivitate liniară. Apelul unei funcții de numeşte simplu, dacă este ultima acțiune din corpul unei funcții. O funcție sau un sistem de funcții având numai apeluri simple, este liniar recursiv repetitiv. Exemplu: *factorialul*. Codul de mai jos reprezintă prototipul unei recursivități liniar repetitive.

```
Tip2 r(Tip1 x) {
       if (b(x)) r((k(x));
       else h(x);
}
```

unde *r* este funcția recursiv liniar repetitivă, *b* este o funcție booleană de parametru *x*, reprezentând condiția de continuare a recursivității, iar *k* şi *h* reprezintă funcții de parametru *x*.

Recursivitate neliniară

În funcțiile recursiv neliniare, două sau mai multe apeluri recursive pot apare în aceeaşi ramificație a unei alternative.

Recursivitate cascadată

În corpul funcției ***f*** pot apare alte apeluri ale lui ***f***, rezultatele acestor apeluri fiind legate de operatori. Exemplu: calcularea numerelor lui Fibonacci.

```
int fibonacci(int n) {
       if (n<=1) return n;
       else return fibonacci(n-2) + fibonacci(n-1);
}
```

Eliminarea recursivității liniare

Forma nerecursivă a unui algoritm este de preferat formei recursive, din punct de vedere al timpului de execuție şi al memoriei ocupate. Avantajul formei recursive îl constituie în primul rând facilitatea şi eleganța

scrierii programelor (de exemplu, comparaţi programul care rezolvă *problema turnurilor din Hanoi* în cele două variante – iterativ şi recursiv). În alegerea căii recursive sau nerecursive (iterative) de rezolvare a unei probleme, programatorul trebuie să stabilească prioritățile în realizarea programului, analizând complexitatea problemei, naturaleţea exprimării, uşurinţa proiectării şi testării programului, eficienţa în execuţie. Astfel, dacă problema e de complexitate redusă, însă se cere eficienţă maximă, se va alege varianta nerecursivă. Varianta recursivă este preferată acolo unde înlocuirea complexității presupune tehnici de programare speciale, algoritmul pierzându-şi naturaleţea.

Algoritmul general de eliminare a recursivității liniare este următorul:

i. Se declară o stivă, care se inițializează ca fiind vidă. Pe acestă stivă urmează să se salveze parametrii formali, şi variabilele locale funcției recursive.
ii. Cât timp condiția de continuare a recursivității e îndeplinită, se efectuează urmatoarele:
    iii. Se salvează pe stivă valorile actuale pentru argumentele funcției recursive şi variabilele locale.
    iv. Se execută instrucțiunile funcției recursive.
    v. Se modifică valorile argumentelor funcției recursive.
vi. Când condiția de continuare nu mai e îndeplinită, dacă stiva nu e goală, se aduce un set de variabile de pe stivă şi se calculează valoarea dorită (dupa apelul funcției recursive) – eventual se execută instrucțiunile de după apelul funcției recursive, apoi se trece la pasul ii.

11.5. EXERCIȚII ŞI PROBLEME

1. La fiecare apel recursiv al unui subprogram, în memoria stivă sunt salvate:
   a) adresa de revenire, valorile variabilelor locale şi a parametrilor transmişi prin referință.
   b) adresa de revenire şi valorile variabilelor globale.

c) adresa de revenire, valorile variabilelor locale şi a parametrilor transmişi prin valoare şi adresele parametrilor transmişi prin referință.
d) adresa de revenire, valorile variabilelor locale şi globale.

2. Care dintre următoarele afirmații sunt corecte ?
a) Programul principal / funcția *main*() (Pascal / C) nu poate conține un autoapel.
b) Un subprogram este recursiv dacă şi numai dacă nu conține mai mult de un autoapel.
c) Orice subprogram recursiv se poate implementa şi iterativ.
d) Un subprogram recursiv trebuie să aibă cel puțin un parametru transmis prin valoare.

3. Se consideră următorul subprogram recursiv:

```
int p(int n, int x)
{
        if(x==n)        return 1;
        else
                if(n%x==0)    return 0;
                else
                        return p(n, x+1);
}
```

În urma apelului $p(n,2)$ funcția va returna valoarea 1 dacă şi numai dacă:
a) numărul natural *n* nu este prim.
b) numărul natural *n* este prim.
c) numărul natural *n* este par.
d) numărul natural *n* este impar.

4. Să se realizeze un subprogram recursiv, care primind ca parametru un număr întreg *n*, să întoarcă printr-un alt parametru cifra sa maximă.

5. Scrieți o funcție recursivă care să calculeze suma *S*, unde *n* va fi transmis ca şi parametru:

$$S = 1 - 2^2 + 3^2 - 4^2 + \ldots \pm n^2$$

6. a) Adresa de revenire dintr-o funcție recursivă este întotdeuna aceeaşi ? De ce DA sau de ce NU ? Argumentați.
b) Pe durata execuției unui program principal o subrutină (nerecursivă) este apelată **exclusiv** printr-o instrucțiune JSR (apel de subrutină –

JAL la MIPS sau CALL la Intel) situată la o anumită adresă în acest program, de 723 de ori. De ce este dificil de predicționat în acest caz instrucțiunea RET (revenirea în programul apelant) de la finele subrutinei apelate?

7. Se consideră următoarea secvență de program C:

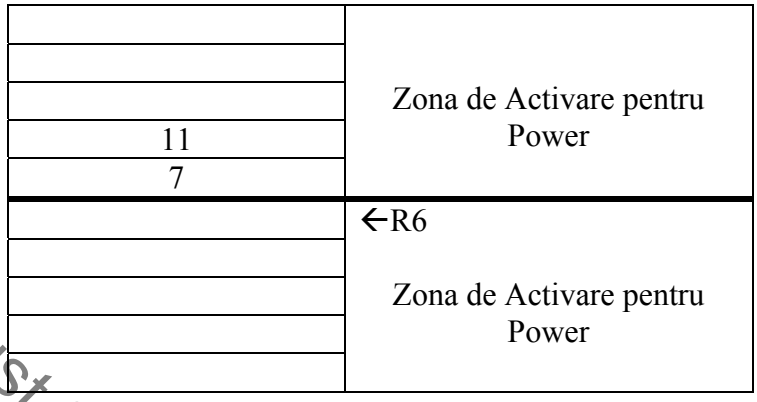
```
#include <stdio.h>
int Power(int a, int b);

int main(void)
{
        int x, y, z;
        printf("Introduceti doua numere: ");
        scanf("%d %d", &x, &y);
        if (x>0 && y>0)
                z = Power(x,y);
        else
                z = 0;
        printf("Rezultatul este %d ", z);
}

int Power(int a, int b)
{
        if (a<b)
                return 0;
        else
                return 1 + Power(a/b,b);
}
```

Se cere:

a) Ce afişează programul dacă se citesc de la tastatură perechile:
   i) 4 9
   ii) 27 5
   iii) -1 3
b) Ce calculează funcția Power() ?
c) Descrieți un instantaneu (o imagine a zonei de activare a stivei de date aferentă funcției Power) în cazul apelului Power(11,7).



8. Se consideră următoarea secvență de program C:

```
int Sigma (int k)
{
        int t;

        t = k-1;
        if(k==0)
                return 0;
        else
                return (k + Sigma(t));
}
```

Se cere:

a) Transformați funcția recursivă într-una iterativă care să realizeze același lucru. Se presupune că sigma este apelată doar cu parametrii pozitivi.
b) Considerând exact 1 KB de memorie contiguă disponibilă exclusiv pentru stiva de date aferentă apelului recursiv al funcție Sigma, şi ştiind că adresele şi numerele întregi sunt pe 16 biți, determinați câte apeluri ale funcției pot fi făcute (valoarea maximă a lui k) pentru a nu obține mesajul *Stack Overflow*. Se consideră că alte variabile temporare nu ocupă spațiul pe stiva de date.

12. POINTERI ŞI TABLOURI. TRANSFERUL PARAMETRILOR PRIN REFERINȚĂ. POINTERI SPRE FUNCȚII

12.1. INTRODUCERE

În acest capitol sunt introduse două construcții de programare simple dar (foarte puternice) extrem de importante în manipularea de către limbajul de nivel înalt (C, C++, Java, Pascal) a conținutului locațiilor de memorie – **pointerii** şi **tablourile**. Pentru început se reaminteşte (vezi capitolul 10) că o variabilă reprezintă o locație de memorie caracterizată prin adresă şi conținut. **Pointerul**[18] **reprezintă adresa unei zone de memorie**; **el face posibil accesul indirect la conținutul respectivei locații**. Din punct de vedere al conținutului memoriei indicate, se disting următoarele categorii de pointeri:

- **Pointeri către** (adrese de) **date** – conțin adresa unei variabile.
- **Pointerii către** (adrese de) **funcții** – conțin adresa codului executabil a unei funcții. Deşi în C, o funcție nu este o variabilă este posibil a se referi pointeri la funcții. Aceştia pot fi atribuiți, plasați în tablouri, transmişi ca parametrii altor funcții.
- **Pointeri generici** (pointeri **void ***) – conțin adresa unui obiect oarecare, de orice tip. Prin declararea **void *** este anunțat compilatorul să nu verifice tipul datei spre care se pointează. Se utilizează atunci când nu se cunoaşte exact tipul de date (adresa şi conținutul) care va fi instanțiat în mod dinamic la execuție.

Înțelegerea şi utilizarea corectă a pointerilor este esențială pentru acuratețea şi eficiența aplicațiilor scrise în limbajele de nivel înalt (C, C++, Java). Trei motive justifică acest lucru:

- **Pointerii oferă posibilitatea de a modifica argumentele de apelare a funcțiilor** (*transferul parametrilor prin referință*).

[18] Termenul de *pointer* a fost preluat în limba română şi poate fi folosit cu sensul de referință, indicator de adresă, localizator.

- **Pointerii permit** (facilitează) rezervarea (alocarea) **dinamică a memoriei** (doar în momentul execuţiei se cunoaşte adresa reală şi conţinutul unei variabile şi nu în etapa de compilare). Acest lucru este foarte important mai ales atunci când anumite variabile alocate static (în faza de compilare) nu sunt utilizate decât pentru foarte scurt timp sau chiar de loc pe parcursul execuţiei aplicaţiei – vezi cazul tablourilor de dimensiuni foarte mari. Alocarea de zone de memorie şi eliberarea lor în timpul execuţiei programelor permite gestionarea optimă a memoriei de către programe. Cu ajutorul pointerilor pot fi create diferite modalităţi de organizare a datelor în structuri, care cresc sau descresc pe durata execuţiei programelor (un exemplu ar fi stiva de date aferentă funcţiilor) [Patt03].
- **Pointerii pot îmbunătăţi eficienţa anumitor rutine**.

Pe lângă aceste facilităţi, pointerii impun însă şi responsabilitate atunci când sunt folosiţi. Pointerii neiniţializaţi dar utilizaţi în operaţii (sau care conţin valori neadecvate) pot determina blocarea sistemului de operare. De asemenea, pointerii folosiţi în mod incorect implică erori greu de depistat.

Întrucât pointerul reprezintă adresa unei zone de memorie rezultă că orice operaţie cu pointeri presupune citirea / scrierea în memorie (iar în cazul celor neiniţializaţi – într-o zonă de memorie necunoscută) [Zah04]. Efectul operaţiei de citire nu este atât de grav din punct de vedere al funcţionalităţii sistemului ci doar strict din punct de vedere al aplicaţiei, care va folosi astfel o valoare eronată. Problema cea mai neplăcută la folosirea pointerilor neiniţializaţi poate apare la atribuirea (asignarea) unei valori conţinutului acestora. Scrierea într-o zonă de memorie necunoscută poate distruge (altera) chiar codul sau datele proprii, sau mai rău, zone necesare sistemului de operare (apeluri sistem, rutine de tratare a diverselor întreruperi hardware, etc). Efectul se va vedea doar mai târziu în timpul execuţiei programului. Exemplul următor ilustrează un caz de pointer neiniţializat.

```
void main(void)
{
        int x, *p;
        x = 12;
        *p = x;
        …
}
```

Această secvență de cod ilustrează o eroare destul de frecventă şi greu de sesizat, care atribuie valoarea 12 unei locații de memorie necunoscute, deoarece pointerul *p* nu a fost inițializat în momentul în care s-a executat instrucțiunea **p = x;*. Pentru aplicații de dimensiuni reduse, deşi pointerul nu este inițializat este probabil ca el să indice spre o adresă „*sigură*" (una care nu intră în zona de cod, date sau aferentă sistemului de operare). Însă, cu cât aplicația este mai mare şi mai complexă (folosirea tablourilor de pointeri sau liste înlănțuite), efectul acestei greşeli poate deveni „*neplăcut*" şi chiar dezastruos. O convenție uzuală în cazul aplicațiilor cu pointeri, respectată de majoritatea programatorilor în C/C++, sugerează ca: „*unui pointer care nu indică efectiv o locație de memorie validă i se dă valoarea NULL (0)*". Astfel, un pointer NULL se consideră că nu indică spre nimic şi nu ar trebui folosit.

O ultimă observație legată de inițializarea pointerilor se referă la cazul în care un pointer ia valoarea altui pointer (de exemplu, int *p, int *q; p=q;). În această situație se pierde legătura cu locația spre care indicase *p*, iar locația spre care referise *q* este acum partajată atât de *p* cât şi de *q*, orice modificare la respectiva locație din partea unuia dintre ei afectându-i pe ambii pointeri (vezi exemplul din figura 12.1).

12.2. SEMNIFICAȚIE ŞI DECLARARE

De regulă, pointerii sunt utilizați pentru a face referire la date cunoscute prin adresele lor. Declarația unui pointer este asemănătoare cu declarația oricărei variabile, singura deosebire fiind aici că, numele pointerului este precedat de caracterul * [Neg97].

Tip **nume_pointer*; /* unde *nume* este un pointer care referă spre o zonă de memorie care conține date de tipul *Tip*.*/

În construcția **nume_pointer* indicată mai sus, caracterul * se consideră ca fiind un operator unar care furnizează valoarea din zona de memorie a cărei adresă este conținută de *nume_pointer*. Operatorul unar * are aceeaşi prioritate ca toți ceilalți operatori unari din C.

În declarația **int *p;** tipul *int* stabileşte faptul că *p* conține adrese de zone de memorie alocate datelor de tip int. Declarația lui *p* poate fi

interpretatată astfel: **p* reprezintă conţinutul zonei de memorie spre care indică p, iar acest conţinut este de tip int. De exemplu, dacă *p* este un pointer care are ca valoare adresa zonei de memorie alocată variabilei întregi a, atunci **p* reprezintă chiar valoarea variabilei *a*. Pentru atribuirea unui pointer *p* a adresei unei variabile (*iniţializarea pointerului*) se va folosi operatorul unar &. De exemplu, declaraţiile **int a; int *p = &a;** exprimă de fâpt trei operaţii: o declaraţie a variabilei **a** (alocare statică a memoriei – **int a;**) o declaraţie a unui pointer spre tipul int (**int *p;**) şi o iniţializare a lui p cu adresa variabilei a (**p = &a;**).

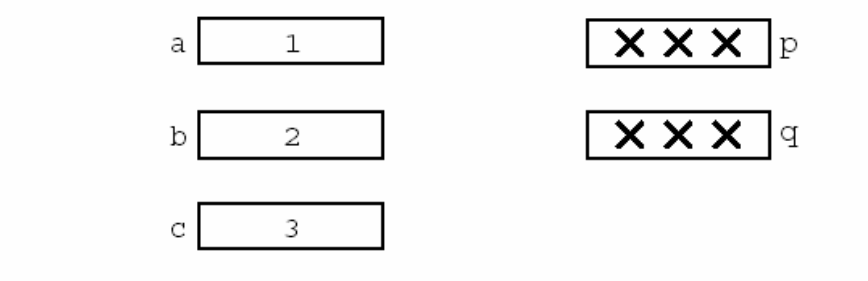
În continuare sunt prezentate câteva echivalenţe de construcţii de limbaj folosind pointeri.

Se consideră declaraţiile: int x, y; int *p;	
y = x + 100;	p = &x; y= *p + 100;
x = y;	p = &x; *p= y;
x++;	p = &x; (*p)++;

Exemplul următor ilustrează (şi grafic) instrucţiunile prin care un pointer este iniţializat şi cum se modifică referinţele la variabile pe parcursul unei secvenţe de program.

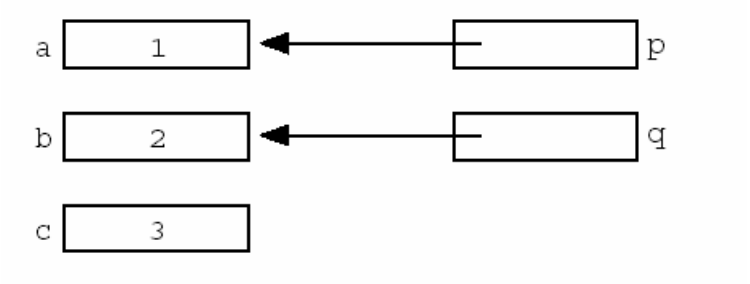
```
void Exemplu_cu_Pointer(){
        int a = 1;
        int b = 2;
        int c = 3;
        int* p;
        int* q;

        // Conţinutul memoriei în acest punct (T1) este următorul (pointerii p
        şi q nu au fost încă iniţializaţi).
```

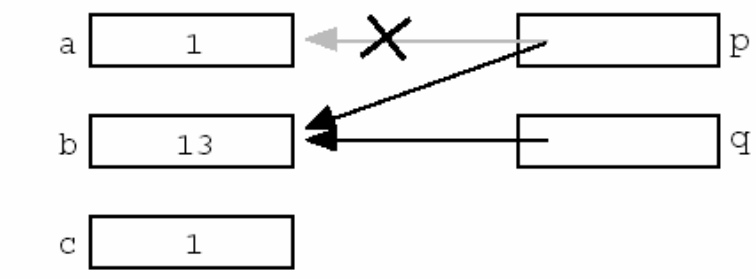


p = &a; // este iniţializat *p* să indice spre variabila *a*, adresa lui a fiind cunoscută în momentul compilării
q = &b; // este iniţializat *q* să indice spre variabila *b*, adresa lui b fiind cunoscută şi ea în momentul compilării

// Conţinutul memoriei în acest punct (T2) este următorul:



c = *p;
p = q; // se pierde referinţa anterioară a lui *p* (care era spre *a*) şi va partaja împreună cu *q* aceeaşi referinţă spre *b*.
*p = 13; // prin modificarea efectuată asupra valorii de la locaţia spre care pointează *p* se modifică şi *q.
// Conţinutul memoriei în acest punct (T3) este următorul:



}

Figura 12.1. Aplicaţie C folosind pointeri

12.3. ALOCAREA ŞI ACCESAREA DE VARIABILE. LEGĂTURA DINTRE NIVELUL HIGH ŞI LOW VĂZUTĂ PRIN INTERMEDIUL MODURILOR DE ADRESARE

În cadrul LC-3 ISA [Patt03] **accesul la conţinutul unei locaţii de memorie corespunzătoare unei variabile alocate static** poate fi făcut prin intermediul instrucţiunii *LD* R_{dest}*, Adresă* (**mod de adresare direct**) – dacă se află într-un spaţiu de (+255 / -256) locaţii faţă de adresa instrucţiunii curente şi respectiv, folosind succesiunea de instrucţiuni (1) *LEA* $R_{bază}$*, Adresă* şi (2) LDR R_{dest}*,* $R_{bază}$*, #0* – cu mod de adresare indexat, mai ales atunci când locaţia de memorie se află într-un spaţiu îndepărtat cu cel puţin 256 de locaţii faţă de PC-ul instrucţiunii curente. **Accesul la conţinutul unei locaţii spre care indică o variabilă de tip *pointer*** (alocare dinamică a memoriei) se face folosind o instrucţiune de acces la memorie în **mod de adresare indirect** LDI R_{dest}*, Adresă* sau, după cum poate fi observat şi în exemplul următor, printr-o succesiune de trei instrucţiuni: (1) *LEA* $R_{bază}$*, Adresă*, (2) LDR R_{cont}*,* $R_{bază}$*, #0* şi (3) LDR R_{dest}*,* R_{cont}*, #0*.

Aplicaţie 1: Accesul la conţinutul unei locaţii spre care indică o variabilă de tip *pointer*.

Se consideră următoarea configuraţie arhitecturală:

Adresă_memorie	**Conţinut_locaţie**
x3050	x70A2
x70A2	x70A3
x70A3	xFFFF
x70A4	x123B

Registrul PC indică valoarea x3010 iar instrucţiunile de la adresele x3010 la x3012 sunt ilustrate mai jos. Să se determine valoarea stocată în R6 după execuţia celor 3 instrucţiuni. Pot fi înlocuite cele trei instrucţiuni cu una singură care să rezolve aceeaşi problemă ? Care este aceasta ?

Adresă_memorie	**Codificare_instrucţiune**
x3010	LEA R3, x3050
x3011	LDR R4, R3, #0
x3012	LDR R6, R4, #0

	Conţinut regiştrii după execuţia instrucţiunii		
Instrucţiune	**R3**	**R4**	**R6**
LEA R3, x3050	x3050		
LDR R4, R3, #0	x3050	x70A2	
LDR R6, R4, #0	x3050	x70A2	x70A3

Practic secvenţa de 3 instrucţiuni realizează: **R6 ← Mem[Mem[x3050]]** care poate fi codificată mai simplu printr-o singură instrucţiune: **LDI R6, x3050**, execuţia însă presupunând acelaşi număr de paşi (3 – determinare adresă, determinare conţinut locaţie de la adresa respectivă care va reprezenta adresa finală de la care se va aduce operandul sau valoarea dorită).

Aplicaţie 2.1: Corespondenţa limbaj de nivel înalt –C şi limbaj asamblare LC-3 folosind pointeri [Patt03].

Cele două secţiuni de cod de mai jos (una în C, cealaltă în asamblare LC-3) trebuie să realizeze acelaşi lucru. Se consideră variabilele x, p şi z declarate global. Instrucţiunile scrise cu litere aldine şi subliniate au fost adăugate pentru a crea corespondenţa corectă C – LC-3.

int x; int *p; int z;		**.**ORIG x3000

main()
{
x = 0;
		LD R0, X
		AND R0, R0, #0
		ST R0, X

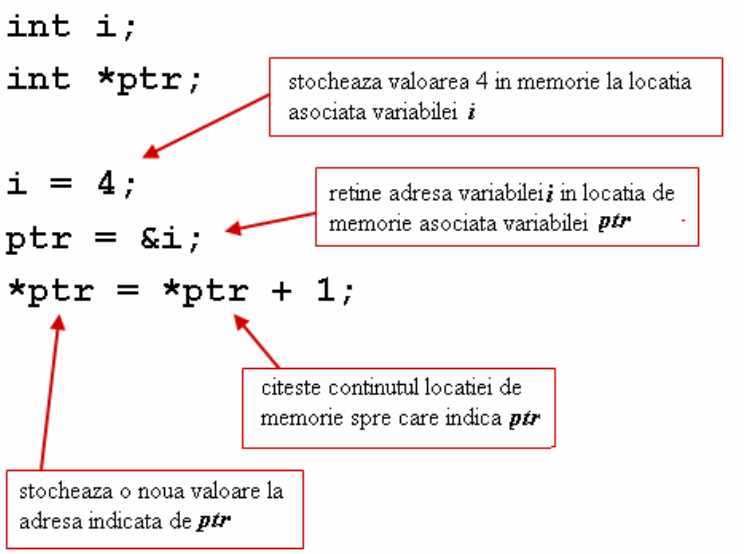
.		.

x++;		LD R0, X
		ADD R0, R0, #1
		ST R0, X

p =&x;		LEA R0, X ST R0, P
z = (*p) + 1;		**LDI R0, P** ADD R0, R0, #1 ST R0, Z HALT
}	X	**.**BLKW 1

	P	.BLKW 1
	Z	.BLKW 1
		.END

Aplicaţie 2.2: Corespondenţa limbaj de nivel înalt –C şi limbaj asamblare LC-3 folosind pointeri. Se consideră de această dată că cele două variabile sunt declarate local funcţiei main().



; ***i*** este prima variabilă declarată local (offset-ul 3 faţă de începutul stivei de date aferentă funcţiei *main()*); ***ptr*** este a doua variabilă (ofset 4).

; **i = 4;**

```
        AND   R0, R0, #0      ; iniţializare R0
        ADD   R0, R0, #4      ; stochează valoarea 4 în R0
        STR   R0, R6, #3      ; scrie conţinutul lui R0 în stiva de date aferentă
                                funcţiei main() la locaţia corespunzătoare
                                variabilei i
```

; **ptr = &i;**

```
        ADD   R0, R6, #3      ; R0 = R6 + 3 (R0 reţine adresa variabilei i)
        STR   R0, R6, #4      ; memorează adresa lui i în ptr
```

; *ptr = *ptr + 1;

```
        LDR  R0, R6, #4       ; R0 = ptr (valoarea adresei)
```

```
LDR  R1, R0, #0     ; încarcă de la ptr (*ptr) în R1
ADD  R1, R1, #1     ;incrementează valoarea lui R1
STR  R1, R0, #0     ;stochează rezultatul la adresa specificată de
                    ptr (acolo unde R0 pointează)
```


12.4. TRANSFERUL PARAMETRILOR PRIN REFERINȚĂ LA APELUL FUNCȚIILOR

Pentru a demonstra necesitatea transferului parametrilor prin adresă în cazul apelurilor de funcții se va folosi următoarea secvență de cod C, în care, funcția *Swap*() dorește să interschimbe valorile celor două argumente. Funcția Swap() este des apelată în cazul aplicațiilor de sortare.

```
#include <stdio.h>
void Swap(int firstVal, int secondVal);

int main()
{
        int valueA = 13;
        int valueB = 333;

        printf("Inaintea apelului functiei Swap ");
        printf("valueA = %d and valueB = %d\n", valueA, valueB);
        Swap(valueA, valueB);
        printf("Dupa apelul funcției Swap ");
        printf("valueA = %d and valueB = %d\n", valueA, valueB);
}

void Swap(int firstVal, int secondVal)
{
        int tempVal;    /* Retine valoarea primului parametru – firstVal la
                           interschimbare */

          tempVal = firstVal;
          firstVal = secondVal;
          secondVal = tempVal;
}
```

Funcţia Swap() este apelată din programul principal main() cu argumentele *valueA* egal cu 13 şi *valueB* egal cu 333. Rezultatul dorit în urma apelului este ca valueA să fie egal cu 333 iar valueB egal cu 13. Cu toate acestea, argumentele transmise funcţiei Swap() rămân nemodificate în urma compilării şi execuţiei codului.

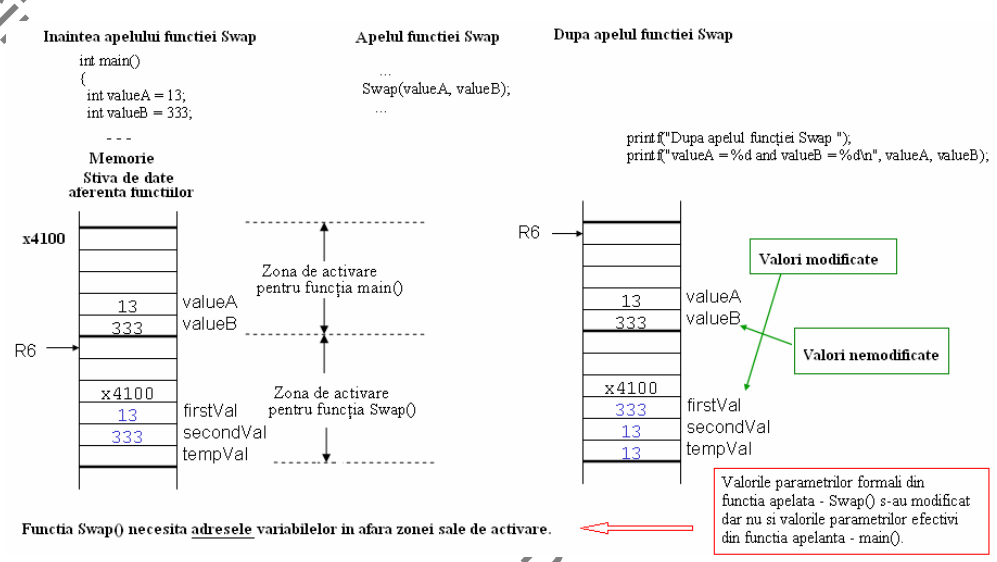


Figura 12.2. Execuţia funcţiei *Swap* – imaginea zonei de activare înainte şi după apel

Examinând stiva de date aferentă funcţiilor *main()* şi *Swap()* pe durata execuţiei programului (vezi figura 12.2) se observă cum interschimbarea operează doar asupra copiilor locale ale parametrilor *firstVal* şi *secondVal*, iar la revenirea din Swap() în main() valorile modificate sunt pierdute odată cu descărcarea de pe stivă a zonei de activare aferentă funcţiei Swap(). „*Vinovat*” de acest fapt este limbajul C care *transmite argumentele de la nivelul funcţiei apelante la cel al funcţiei apelate prin* ***valoare***. „*C*”-ul evaluează fiecare argument care apare în apelul de funcţie ca o expresie şi plasează **valoarea** acesteia în locaţia corespunzătoare din zona de activare aferentă funcţiei apelate. Pentru ca funcţia Swap() să poată modifica argumentele actuale – parametri efectivi pe care funcţia apelantă main() îi transmite, ea (Swap) trebuie să aibă acces la zona de activare a funcţiei apelante (trebuie să acceseze locaţiile la care sunt stocate argumentele pentru a modifica valorile acestora). Rezultă, în mod evident că, funcţia Swap are nevoie de adresele variabilelor valueA şi valueB din main() pentru a modifica valorile reţinute de acestea. După cum poate fi observat din

secvența de cod de mai jos dar şi din figura 12.3, acest lucru poate fi realizat folosind pointerii şi operatorii asociați de indirectare – * respectiv adresă – &.

```
#include <stdio.h>
void NewSwap(int *firstVal, int *secondVal);

int main()
{
      int valueA = 13;
      int valueB = 333;

      printf("Inaintea apelului functiei NewSwap ");
      printf("valueA = %d and valueB = %d\n", valueA, valueB);
      NewSwap(&valueA, &valueB);
      printf("Dupa apelul functiei NewSwap ");
      printf("valueA = %d and valueB = %d\n", valueA, valueB);
}

void NewSwap(int *firstVal, int *secondVal)
{     /* Parametri formali sunt pointeri la numere întregi. Funcția
         apelantă transmite adresele variabilelor pe care funcția vrea să le
         interschimbe. */
         int tempVal;      /* Retine valoarea primului parametru – firstVal
                              la interschimbare */

          tempVal = *firstVal;
          *firstVal = *secondVal;
          *secondVal = tempVal;
}
```

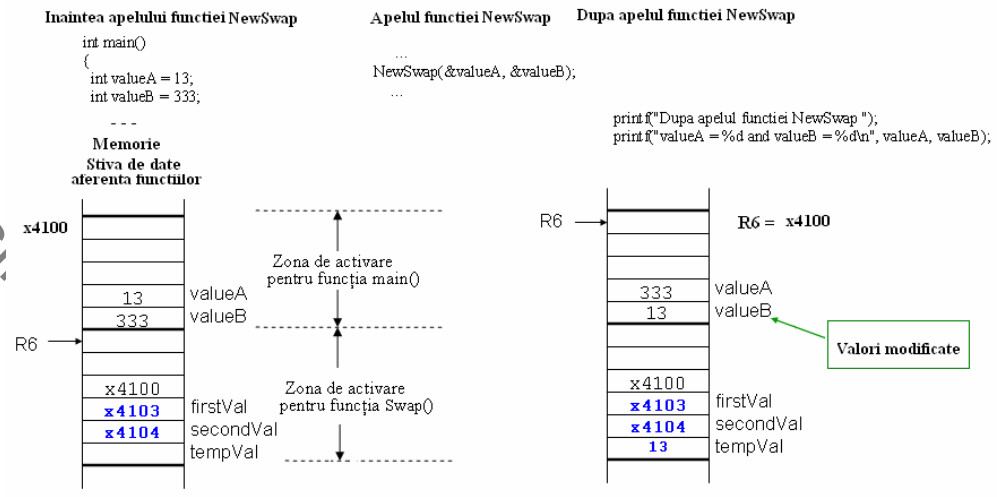


Figura 12.3. Execuţia funcţiei *NewSwap* – imaginea zonei de activare înainte şi după apel

În continuare, sunt ilustrate schimbările intervenite la nivelul funcţiei Swap(), de la începutul acestui paragraf (12.4) şi din figura 12.2, pentru a realiza obiectivul iniţial, cel de interschimbare a argumentelor primite. Noua variantă (revizuită) a functiei Swap() se va numi NewSwap(). Prima modificare constă în faptul că parametrii formali ai funcţiei NewSwap nu mai sunt de tip întreg (int) ci pointeri la întregi (int *). În noua funcţie, cei doi parametri sunt adresele de memorie a celor două variabile care se vor inversate (interschimbate).Operatorul de indirectare folosit în prototipul funcţiei NewSwap semnifică faptul că variabilele spre care se pointează vor fi interschimbate. Întrucât aceste valori se regăsesc în zona de activare a funcţiei apelante (*main*), la apelul funcţiei NewSwap se va folosi operatorul adresă (vezi figura 12.3).

Echivalenţa C – LC-3 la nivelul funcţiei apelante main() şi la nivelul funcţiei apelate NewSwap() [Patt03].

Funcţia main() urmăreşte interschimbarea valorilor variabilelor locale valueA şi valueB, motiv pentru care transmite funcţiei NewSwap() adresele celor două variabile.

NewSwap(&valueA, & valueB);

Codul asamblare LC-3 care realizează acest lucru este următorul:

ADD R0, R6, #3; în R0 se calculează adresa variabilei *valueA* – unde R6 indică spre adresa primei locații din stiva de date aferentă funcției main().

STR R0, R6, #8 ; se memorează această adresă (R0) în locația corespunzătoare primului parametru efectiv transmis funcției NewSwap() – (vezi figura 12.3).

ADD R0, R6, #4; în R0 se calculează adresa variabilei *valueB*.

STR R0, R6, #9 ; se memorează această adresă (R0) în locația corespunzătoare celui de-al doilea parametru efectiv transmis funcției NewSwap() – (vezi figura 12.3).

Secvența de cod asamblare la nivelul funcției apelate NewSwap() este următorul:

; int tempVal = *firstVal;

LDR R0, R6, #3; în R0 se încarcă valoarea parametrului formal, care reprezintă **adresa variabilei *valueA*** din funcția apelantă (R0←x4103). În acest moment, R6 indică spre adresa primei locații din stiva de date aferentă funcției NewSwap().

LDR R1, R0, #0 ; în R1 se încarcă **valoarea variabilei *valueA*** (R1←13).

STR R1, R6, #5; se salvează în variabila tempVal conținutul lui R1 (tempVal←13).

; *firstVal = *secondVal;

LDR R1, R6, #4; în R1 se încarcă valoarea parametrului formal, care reprezintă **adresa variabilei *valueB*** din funcția apelantă (R1←x4104). De asemenea, R6 indică spre adresa primei locații din stiva de date aferentă funcției NewSwap().

LDR R2, R1, #0 ; în R2 se încarcă **valoarea variabilei *valueB*** (R1←333).

STR R2, R0, #0 ; Se modifică prima variabilă cu valoarea finală (valueA=333)

; *secondVal = tempVal;

LDR R2, R6, #5 în R2 se încarcă **valoarea variabilei locale tempVal** din funcția apelată (R2←13).

STR R2, R1, #0 ; Se modifică a doua variabilă cu valoarea finală (valueB=13).

Diferenţa esenţială între cele două tipuri de apeluri, prin valoare respectiv prin referinţă, este următoarea: **funcţia apelată prin valoare nu poate modifica parametrii efectivi (actuali) din funcţia care a făcut apelul neavând acces la ei, în schimb, în cazul apelului prin referinţă, funcţia apelată, dispunând de adresa parametrilor efectivi, îi poate modifica pe aceştia** [Patt03, Neg97]**.**

12.5. POINTERI SPRE FUNCŢII

O caracteristică generatoare de confuzii dar performantă a limbajului C este *pointerul către o funcţie*. Deşi o funcţie nu este o variabilă, ea are o localizare în memorie care poate fi atribuită unui pointer. Adresa unei funcţii este punctul de intrare în funcţie. Astfel, un pointer către funcţie poate fi utilizat pentru a apela respectiva funcţie.

În continuare se reaminteşte pe scurt modul în care o funcţie este compilată şi apelată (vezi capitolul 10). Codul sursă este transformat în cod obiect (tabela de simboluri, analiză şi sinteză) şi se stabileşte un punct de intrare – vezi directiva .ORIG (la LC-3) [Patt03] care stabileşte valoarea cu care se încarcă PC-ul primei instrucţiuni din funcţie. În timpul rulării programului, atunci când este apelată o funcţie, acest punct de inserare este apelat de limbajul maşină (inserat pe stiva de date aferentă programului apelant – vezi figura *11.8. Stiva de date asociată funcţiei Fib(n)*). Dacă un pointer conţine adresa punctului de intrare, poate fi folosit pentru a apela acea funcţie. Adresa unei funcţii se obţine utilizând numele unei funcţii fără nici o paranteză sau argumente.

Pentru a declara o variabilă pointer către o funcţie, numele pointerului trebuie precedat de simbolul *; de exemplu, **int (*pf)(long)** este un pointer către o funcţie care are un parametru de tip long şi întoarce un întreg. Practic, se declară o variabilă de tip pointer, care este capabilă să reţină un pointer către o funcţie ce returnează un întreg. Considerând o funcţie care determină numărul de cifre al unui număr întreg primit ca parametru – **int cif_control(long)**; rezultă următoarea asignare corectă [Zah04, Neg97]:

pf = cif_control;

Programul principal va conţine următoarele linii de cod:

```
int rez, n;
cin>>n;
pf = cif_control;
```

rez = (*pf)(n);

Pot apărea însă şi următoarele erori:

a) ***pf = cif_control(n);*** unde pf este pointer la funcție iar cif_control(n) este un întreg. Mesajul de eroare este următorul: „*Cannot convert ‚int' to ‚int (*)(long)'* ".

b) ***pf = &cif_control(n);*** nu se poate obține adresa rezultatului unei funcții deoarece transferul rezultatului se face prin stivă. Mesajul de eroare este următorul: „*Must take address of a memory location*".

```
#include<iostream.h>
int cif_control(long);
int (*pf)(long);
void main()
{
        long n;
        int rez;

        cout<<"n=";
        cin>>n;
        pf = cif_control;
        rez = (*pf)(n);
        cout<<rez;
}

int cif_control(long a)
{
        int z=0;
        while (a!=0)
        {
            z++;
            a=a/10;
        }
        return z;
}
```


Întrucât numele unei funcții este un pointer spre funcția respectivă el poate fi folosit ca parametru efectiv la apelurile de funcții. Un exemplu larg răspândit în care este nevoie de un astfel de transfer este cel privitor la calculul aproximativ al integralelor definite prin metoda dreptunghiului sau

cea a *trapezului*. În continuare se reaminteşte pe scurt aparatul matematic utilizat [Neg97].

Dându-se o funcţie continuă pe un interval închis [a, b], f: [a, b] → R, atunci, integrala definită $\int_a^b f(x)dx$ este aria trapezului curbiliniu, determinat de axa Ox, dreptele y=a şi y=b şi graficul funcţiei f(x) pe intervalul [a, b]. Aproximarea ariei unui trapez curbiliniu este mult mai eficientă în cazul când pe fiecare din segmentele elementare este aproximată printr-un trapez şi nu printr-un dreptunghi. Intervalul de calcul al integralei [a, b] este divizat în n intervale de lungime egale $h = \frac{b-a}{n}$. Rezultă astfel un set de puncte elementare (x_i = a + i*h, cu i = 0, …,n.) în care se va calcula valoarea funcţiei. Pe un segment elementar [x_i, x_{i+1}] trapezul este determinat de extremităţile segmentului pe axa Ox – punctele (x_i,0) şi (x_{i+1},0) şi de valoarea funcţiei f(x) în extremităţi – punctele (x_i,f(x_i)) şi (x_{i+1},f(x_{i+1})). Reamintind că aria unui trapez (dreptunghic în cazul nostru) este egală cu semi-suma bazelor înmulţită cu înălţimea trapezului rezultă:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} h \cdot \frac{f(x_{i+1})+f(x_i)}{2} = h \cdot \sum_{i=0}^{n-1} \frac{f(x_{i+1})+f(x_i)}{2} \; (= \text{notata de autor } I\_n)$$

Pentru a calcula integrala definită dintr-o anumită funcţie (în exemplul nostru am considerat $\frac{1}{1+x^2}$ şi respectiv $\frac{1}{1+x}$) cu o precizie de 10^{-7} se vor urma paşii algoritmului.

Pasul **1**: Se alege o valoare iniţială pentru n (numărul de diviziuni). Cu cât n este mai mare va avea loc o divizare mai fină a intervalului [a,b] – o apropiere mai mare de valoarea reală a integralei.

Pasul **2**: Se calculează I_n (valoarea integralei pentru n subintervale).

Pasul **3**: Se calculează I_2n (valoarea integralei pentru 2*n subintervale).

Pasul **4**: Dacă $|I\_n - I\_2n| < 10^{-7}$, algoritmul se întrerupe şi valoarea integralei, cu precizia solicitată, este I_2n, altfel se dublează n, se păstrează ultima valoare a integralei I_n = I_2n şi se reia pasul 3.

În continuare se construieşte o funcţie care calculează valoarea lui I_n – numită **integrala_trapez** [Neg97]. Se impun însă unele observaţii:

- Funcţia f(x) trebuie să figureze printre parametrii funcţiei **integrala_trapez** deoarece calculul integralei definite poate fi aplicat oricărei funcţii. De asemenea, limitele de integrare şi numărul de diviziuni în care se împarte intervalul trebuie transmişi ca şi parametrii.

- Funcţia **integrala_trapez** returnează valoarea aproximativă a integralei şi se va apela printr-o expresie de atribuire de genul: *i_n = integrala_trapez(a,b,n,**f**);* în exemplul nostru s-a considerat *f* fiind funcţia *arctan_deriv* (derivata funcţiei arctg*x*).
- Întrucât funcţia f(x) returnează o valoare flotantă în dublă precizie rezultă că şi integrala (funcţia **integrala_trapez**) va returna acelaşi tip de dată. Aşadar, prototipul funcţiei **integrala_trapez** este următorul: *double **integrala_trapez**(double, double, long , double (*)(double));*
- Evident că, înaintea apelului funcţiei **integrala_trapez** funcţia f(x) trebuie să fie definită sau să fie prezent prototipul ei.
- Construcţia sintactică double (*p)(double) din antetul funcţiei **integrala_trapez** se interpretează astfel:
  - **p* – semnifică faptul că *p* este un pointer.
  - (**p*)(*double*) – înseamnă că *p* este un pointer spre o funcţie cu parametru de tip *double*.
  - *double* (**p*)(*double*) – specifică faptul că *p* este un pointer spre o funcţie care returnează o valoare flotantă în dublă precizie.
- Este necesară includerea lui **p* între paranteze rotunde, deoarece construcţia *double *p*(*double*) este corectă din punct de vedere sintactic, dar înseamnă altceva, parantezele rotunde fiind prioritare operatorului unar *. În acest caz, se declară *p* ca o funcţie cu parametru double şi care returnează un pointer spre o valoare flotantă în dublă precizie. Astfel de declaraţii se întâlnesc des în aplicaţii cu structuri de date cu legături (sau autoreferire) – liste, arbori, etc.
- În momentul definirii funcţiei **integrala_trapez**, necunoscându-se numele concret al funcţiei f(x), ci doar pointerul spre ea, în apelurile f(a), f(a+h), etc, se va înlocui numele funcţiei f(x) prin *p astfel: (*p)(a), (*p)(a+h), etc.

```
#define  a  0.0          /* capătul stâng al intervalului pe care se
                            calculează integrala   */
#define  b  1.0          /* capătul drept al intervalului */
#define  N  10           /* numărul de subdiviziuni al intervalului [a,b]
                            utilizat in calculul integralei */
#define        EPSILON 1e-7/*      precizia de calcul a integralei */
#include <math.h>
#include <stdio.h>
#include <conio.h>
double arctan_deriv(double);        /*      1/(1+x_patrat) */
double integrala_trapez(double, double, long , double (* )(double));
```

```
void main(void)     /* calculeaza integrala din arctan_deriv(x)=1/(1+x*x)
                       in intervalul [a,b] cu o eroare mai mica decat
                       EPSILON */
{
        clrscr();
        long n = N;
        double i_n, i_2n, vabs;
        i_n = integrala_trapez(a,b,n,arctan_deriv);
        do{
                n = 2*n;
                i_2n =integrala_trapez(a,b,n,arctan_deriv);
                if((vabs=i_n - i_2n)<0)
                        vabs = - vabs;
                i_n = i_2n;
        }while(vabs>=EPSILON);
        printf("Valoarea integralei este: %.10g\n",i_2n);
        printf("Integrala se obtine cu precizia %g dupa %ld
                iteratii!\n",EPSILON,n);
        getch();
}

double integrala_trapez(double x, double y, long m, double (*p)(double))
{       /* calculul integralei prin metoda trapezelor - divizarea intervalului
        [a,b] in m subintervale si calculul ariei folosind formula trapezului
        dreptunghic */

        double h,s;
        int i;

        h = (y-x)/m;
        for(i=1,s=0.0;i<m;i++)
                s+=(*p)(x+i*h);
        s+=((*p)(x)+(*p)(y))/2;
        s=s*h;
        return s;
}

double arctan_deriv(double x)
{
        return 1/(1+x*x);
//      return 1/(x+1);
}
```

În ciuda multiplelor avantaje la nivelul limbajului *high*, pointerii introduc un nivel de indirectare (acces suplimentar la memorie) iar la nivel *low*, apelurile de funcţii indirect prin pointer generează salturi / apeluri indirecte, foarte dificil de prezis, cu implicaţii defavorabile asupra performanţei globale de procesare [Flo05]**.**

În continuare este ilustrat un exemplu care justifică afirmaţia că "**funcţiile de bibliotecă determină la nivelul procesorului salturi / apeluri indirecte** (*new*, *qsort*, *scan*, *printf*)". Este vorba de programul de test **qsort**[**.c,.s**], preluat din help-ul oferit de mediul BorlandC şi care sortează un tablou de şiruri de caractere prin intermediul funcţiei de bibliotecă *qsort* care primeşte ca parametri adresa tabloului, numărul de elemente, dimensiunea fiecărui element şi o funcţie de comparare de două şiruri de caractere. Practic o sursă a celor 11 apeluri indirecte o constituie şi **apelul indirect prin pointer la funcţia de comparare** – pas realizat în cadrul funcţiei qsort (*precompilate*) [Flo05].

În continuare se exemplifică pe baza descrierii funcţiei din fişierul **...\BorlandC\Crtl\Clib\qsort.cas**.

Nume funcţie **qsort** – sortează un tablou de elemente pe baza metodei de sortare rapidă "selecţia *elementului median din trei*", prin apelul repetat al unui pointer la o funcţie definită de utilizator (**(*fcmp)()**).

Utilizare **void qsort(**void *base, int nelem, int width, **int (*fcmp)())**;

Prototipul funcţiei se află în **stdlib.h**.

**fcmp* *funcţie de comparare* care acceptă două argumente: *elem1* şi *elem2*, adresele a două elemente ale tabloului de sortat. Rezultatul returnat este următorul:

Dacă elementele sunt în relaţia	*fcmp* returnează
*elem1 < *elem2	Un întreg negativ (elem1 se va afla în tabloul sortat înaintea lui elem2)
*elem1 == *elem2	0
*elem1 > *elem2	Un întreg pozitiv (>0)

Qsort nu returnează nimic.

În continuare se vor prezenta comparativ, exemplificându-se la nivel de instrucţiune, secvenţe din codul sursă **.c** (de nivel înalt) versus **.s** (asamblare) al aplicaţiei **qsort.[c,s]**.

**********Secvenţă de cod din **qsort.c**********

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char list[5][4] = { "cat", "car", "cab",
"cap", "can"};

int sort_function( const void *a,
const void *b)
{
  return( strcmp((char *)a,(char *)b));
}
```

```
int main(void)
{
  int  x;

  qsort((void *)list, 5,
sizeof(list[0]), sort_function);
  for (x = 0; x < 5; x++)
    printf("%s\n", list[x]);
  return 0;
}
```

**********Secvenţă de cod din **qsort.s**********

```
        .globl  list
        .data
        .align  2
list:
        .ascii  "cat\000"               # stocarea în zona de date a adreselor de mesaje:
                                        elementele de tablou care vor fi sortate.

        .ascii  "car\000"
        .ascii  "cab\000"
        .ascii  "cap\000"
        .ascii  "can\000"
        ...
        .text
        .align  2
        .globl  main
        .align  2
        .globl  sort_function__FPCvT0   # adresa funcţiei de sortare
        .text
        .ent    main
main:
        subu    $sp,$sp,32              # alocare spaţiu pe stivă
        sw      $31,28($sp)             # salvare # adresa de revenire în programul apelant -
                                        $31 (sistemul de operare)

        sw      $fp,24($sp)
        move    $fp,$sp
        jal     __main
        la      $4, list                # pregătire parametrii de apel  $4 – adresa tabloului de
                                        sortat
        li      $5,0x00000005           # $5 <- 5 = numărul de elemente al tabloului
        li      $6,0x00000004           # $6 <-4 = dimensiunea în octeţi a fiecărui element de
                                        tablou
        la      $7,sort_function__FPCvT0  # $7 <- adresa funcţiei folosită de către qsort în apelul său
```

```
        jal     qsort                   # la apelul unei funcţii primii 4 parametrii de apel sunt
                                        salvaţi în regiştrii $4 ÷$7 restul, dacă e
                                        # cazul sunt depuşi pe stivă [Flor03]. Funcţia qsort este
                                        precompilată.
        sw      $0,16($fp)
        ...
$L36:
        move    $sp,$fp                 # sp not trusted here
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addu    $sp,$sp,32
        j       $31
        .end    main
        .ent    sort_function__FPCvT0
sort_function__FPCvT0:
        subu    $sp,$sp,24
        sw      $31,20($sp)
        sw      $fp,16($sp)
        move    $fp,$sp
        sw      $4,24($fp)              # salvare pe stivă a parametrilor de apel în cadrul rutinei
                                        apelate indirect prin qsort – adresa şi
        sw      $5,28($fp)              # numărul de elemente al tabloului
        lw      $4,24($fp)
        lw      $5,28($fp)
        jal     strcmp                  # apel funcţie de bibliotecă pentru compararea a două
                                        şiruri de caractere
        move    $3,$2
        move    $2,$3
        ...
$L41:
        move    $sp,$fp
        lw      $31,20($sp)
        lw      $fp,16($sp)
        addu    $sp,$sp,24
        j       $31
        .end    sort_function__FPCvT0
```

Codul asamblare nu prezintă nici un apel indirect de funcţii de bibliotecă (*jal qsort*, *jal strcmp*). La analiza codului obiect însă (după link-editarea sursei cu bibliotecile în cauză) au rezultat **11** apeluri statice indirecte care au generat **61** apeluri dinamice indirecte.

12.6. TABLOURI. RELAŢIA DINTRE POINTERI ŞI TABLOURI

Tabloul unidimensional reprezintă o zonă **contiguă** de memorie care stochează elemente de acelaşi tip şi este alocată static în zona (segmentul la procesoarele Intel) de date, fapt ce implică necesitatea cunoaşterii numărului

de elemente al tabloului în momentul compilării. Spaţiul alocat pentru tablou este egal cu produsul dintre numărul de elemente şi dimensiunea tipului de dată corespunzător fiecărui element component. Datorită faptului că în C/C++ numele tabloului reţine chiar adresa primului element, indicii tabloului vor începe de la zero (spre deosebire de limbajul Pascal unde indicii încep de la unu). Exemple de tablouri: lista numerelor de telefon, alfabetul, etc. Expresia *a[4]* referă al 5-lea element al tabloului *a*.

În C, exista o strânsa legatura între pointeri si tablouri. Când se foloseste numele unui tablou (fără indici), el reprezintă un pointer către primul element al tabloului. Astfel, în C se pot transmite tablouri ca parametri în funcţii (se transmite adresa de început a tabloului). Alt avantaj al tratării unitare pointer-tablou este aplicarea aritmeticii pointerilor pentru accesarea elementelor unui tablou. Identitatea pointer-tablou se observă cel mai bine în operaţiile cu şiruri de caractere, unde şirul de caractere este un tablou de caractere sau un pointer la caracter.

În continuare sunt prezentate aspecte ale echivalenţei pointer – tablou. Se consideră declaraţiile:

```
char cuvant[5];
char *tablou_ptr;
```

De exemplu atribuirea *tablou_ptr=cuvant* este corectă din punct de vedere sintactic întrucât *tablou_ptr* pointează la cuvant[0], deci spre începutul zonei de memorie unde este stocat tabloul de caractere (echivalentă de asemenea cu *tablou_ptr=&cuvant[0]*;). Pornind de la declaraţiile de mai sus, în tabelul următor, pe fiecare linie, expresiile date sunt echivalente:

Echivalenţe pointer – tablou		
tablou_ptr	cuvant	&cuvant[0]
(tablou_ptr + n)	cuvant + n	&cuvant[n]
*tablou_ptr	*cuvant	cuvant[0]
*(tablou_ptr + n)	*(cuvant + n)	cuvant[n]

Pentru a accesa elementele tabloului cuvant[5] (şirul de caractere), se incrementează valoarea pointerului, astfel dacă **tablou_ptr=cuvant**; atunci ***(tablou_ptr++)** va returna valoarea lui *cuvant[0]*, după care *tablou_ptr* va indica spre *cuvant[1]*. Trebuie avut în vedere că expresia *(tablou_ptr++) este diferită de **(*tablou_ptr)++**, care va incrementa valoarea spre care indică *tablou_ptr*, returnând astfel caracterul de cod ASCII (*tablou[0]*+1*)*.

Există însă şi diferenţe între pointeri şi tablouri. Dacă un pointer poate fi incrementat, el indicând spre altă locaţie, nu acelaşi lucru se poate face cu un tablou (adresa sa este cunoscută la compilare şi nu se poate modifica pe parcursul execuţiei programului). Astfel, este posibilă expresia *tablou_ptr=tablou_ptr*+1, dar *cuvant* va fi tot timpul un pointer constant pe durata execuţiei programului.

Aplicaţie 3: Relaţia dintre pointeri şi tablouri.

```
#include <stdio.h>
void main(void)
{
 int t[4]={0, 1, 2,3};
 int *p=&t[1];          /* se declară p de tip pointer la int şi apoi se
                           iniţializează cu adresa lui t[1] – al doilea
                           element al tabloului t */

 printf("%d\n", *p++);  /* Operatorul unar * are prioritate în faţa
                           operatorului de postincrementare ++ => se
                           afiseaza valoarea lui t[1] şi adresa lui p este
                           incrementată pentru a indica pe următorul
                           element al tabloului, adica pe t[2] */
 printf("%d\n", *++p);  /* Este incrementată adresa, deci p va pointa pe
                           t[3] => se va afisa valoarea lui t[3] */
 printf("%d\n", ++*p);  /* afiseaza valoarea incrementata a lui t[3].
                           Incrementarea are loc asupra continutului
                           locatiei de memorie si nu asupra adresei */
}
```

Aplicaţie 4: Echivalenţa C – LC-3 în cazul aplicaţiilor cu tablouri.

```
int main()
{
        int x, grid[10];
        x=grid[3]+1;
        grid[6]=5;
}
```

, x=grid[3]+1;

```
ADD R0, R6, #4 ;    Adresa de bază a tabloului (numele acestuia) de pe
                    stiva de date aferentă funcției main() este scrisă în R0
ADD R1, R0, #3 ;    Se calculează adresa elementului grid[3] şi se depune
                    în R1
LDR R2, R1, #0 ;    Se introduce în R2 valoarea elementului grid[3]
ADD R2,R2,#1;       Se incrementează valoarea lui R2 (grid[3] + 1);
STR R2,R6,#3;       Se introduce noua valoare a lui R2 în locația
                    corespunzătoare lui x pe stiva de date aferentă
                    funcției main (x=grid[3]+1);

; grid[6]=5;
AND R2,R2,#0;       Inițializare R2=0
ADD R2,R2,#5;       R2=5
ADD R0,R6,#4 ;      Adresa de bază a tabloului (numele acestuia) de pe
                    stiva de date aferentă funcției main() este scrisă în R0
ADD R1,R0,#6 ;      Se calculează adresa elementului grid[6] şi se depune
                    în R1
STR R2,R1,#0 ;      Se scrie valoarea 5 din R2 la locația grid[6]
```

12.6.1. FUNCŢII CU PARAMETRI DE TIP TABLOU (VECTOR)

Întrucât transmiterea tablourilor (vectorilor) ca parametrii funcţiilor este un lucru extrem de util în aplicaţii, există două posibilităţi de a-l realiza [Patt03]:

a) transferul fiecărui element al vectorului (dezavantajos din punct de vedere al spaţiului ocupat pe stiva de date aferentă funcţiei apelate, dar şi din punct de vedere al timpului de execuţie – mai ales în cazul tablourilor cu număr foarte mare de elemente).
b) transmiterea unei referinţe la tablou (numele acestuia).

În cazul apelurilor de funcţie în care parametrul efectiv este numele unui tablou, apelul se face prin referinţă. În acest caz, se transferă valoarea numelui tabloului, adică adresa primului său element. Dispunând de adresa de început a tabloului utilizat ca parametru, funcţia apelată poate modifica elementele tabloului respectiv.

În continuare se prezintă un exemplu de funcție având ca parametru un tablou. Programul principal citeşte de la tastatură un tablou de numere întregi şi apelează o funcție care, având ca argument numele respectivului tablou, deteremină media aritmetică a elementelor din tablou.

```
#include <stdio.h>
#define MAX_NUMS 10

int Average(int input_values[]);

int main()
{
 int index;                 /* Variabilă contor folosită în bucla for  */
 int mean;                  /* media aritmetică a numerelor       */
 int numbers[MAX_NUMS];  /* Declarația inițială a tabloului de numere */

 /* Introducerea numerelor de la tastatură */
 printf("Enter %d numbers.\n", MAX_NUMS);
 for (index = 0; index < MAX_NUMS; index++) {
  printf("Input number %d : ", index);
  scanf("%d", &numbers[index]);
 }
 mean = Average(numbers);
 printf("The average of these numbers is %d\n", mean);
}

int Average(int inputValues[])
{
 int index;
 int sum = 0;
 for (index = 0; index < MAX_NUMS; index++) {
       sum = sum + inputValues[index];
 }
 return (sum / MAX_NUMS);
}
```

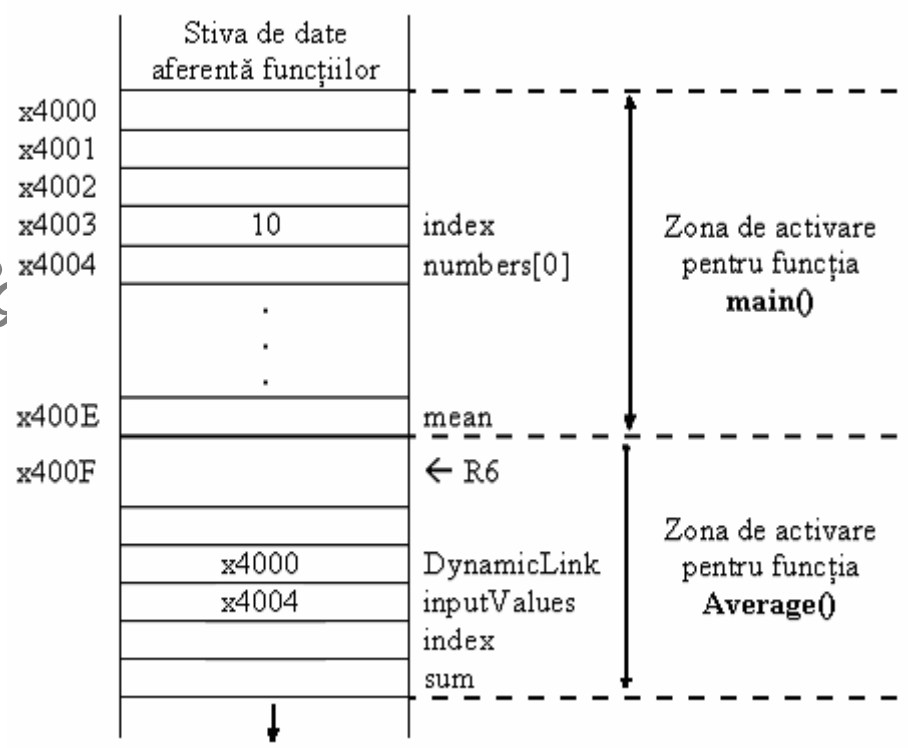


Figura 12.4. Imaginea zonei de activare imediat după apelul funcţiei Average()

12.7. EXERCIŢII ŞI PROBLEME

1. Ce tipăreşte următoarea secvenţă:

```
int m[7]={3,6,4,2,8,9,0}, *p, *q; p=&m[3]; q=++p-2;
cout<<*q<<" "<<*(p+2)<<" "<<*p<<" "<<p-q;
```

a. 4 0 8 2 **b.** 2 2 3 –2 **c.** 8 4 3 0 **d.** 4 0 8 –2 **e.** 2 4 3 4

2. Fie declaraţiile int x[10],*p; Care din următoarele atribuiri este corectă ?

a. x=p; **b.** x[10]=p; **c.** p=x[10]; **d.** p=*x[10]; **e.** p=x;

3. Cunoscând codul ASCII a literei 'A'=65, ce afişează secvenţa:

int x=2; char c='A'; c=c+x; cout<<c;

a. 67 **b.** C **c.** nici o variantă corectă **d.** c **e.** eroare: în C nu se poate aduna un caracter cu un număr.

4. Fie declaraţiile void f(int a, float &b){a++;b=b+1;} Ce va afişa următorul program:
void main() {int x=1; float y=1.1; f(x,y); cout<<x<<' '<<y;}
a. 2 2.1 **b.** 1 2.1 **c.** apel eronat **d.** sintaxă greşită **e.** 1 1.1

5. Ce va afişa programul următor:

```
#include<stdio.h>
void main(void)
{
    int *p,*q;

    p = new (int);
    q = new (int);
    *p = 7;
    *q = *p;
    if(p==q)
        (*p)++;
    else
        (*q)++;
    printf("%d , %d",*p,*q);
}
```

Care este rezultatul afişat de programul anterior dacă instrucţiunea if se înlocuieşte cu if (*p== *q)?

6. Se consideră următoarea secvenţă de cod:

```
#include<stdio.h>
int z,t,*p;
void main(void)
{
    z = 7;
    p = &z;
    p++;
    *p = 19;
    *--p = 24;
```

```
    printf("z= %d p= 0x%x continutul locatiei spre care pointeaza p
    (*p)=%d t=%d",z,p,*p,t);
}
```

Considerând că registrul R5 (pointerul spre începutul zonei de date globale are valoarea x4000) să se ilustreze conţinutul locaţiilor x4000, x4001, x4002 şi x4003 după execuţia fiecărei instrucţiuni din program. Determinaţi ce se afişează pe ecran la finalul execuţiei programului ?

Adresă memorie	**Conţinut**
x4000	
x4001	
x4002	
x4003	

7. Ce afişează următoarele secvenţe de cod C++:

a)

```
#include<iostream.h>

void main()
{
      char v[]={'a','b','c','\0'};
      char *p=&v[0];

      while (*p)
      {
         cout<<p<<" ";
         p++;
      }
}
```

b)

```
#include<iostream.h>

int a=1,c=8,*p;

void main()
{
      p=&a;
      a=*p++;
      cout<<a<<' '<<*p<<endl;
      a=(*p)++;
      cout<<a<<' '<<*p;
}
```

Ce va afişa programul în cazul secvenţei *b* dacă declaraţiile variabilelor se fac local funcţiei main() şi nu global ? Ce modificări trebuiesc aduse programului pentru a afişa acelaşi lucru ca în cazul iniţial (cu declaraţiile globale). Indicaţie: pe un sistem cu procesor Intel, organizarea memoriei presupune că stiva de date aferentă funcţiilor creşte (în conţinut şi capacitate) de la adrese mari spre adrese mici (variabilele succesive sunt stocate la adrese care descresc).

8. Translataţi din C în limbaj de asamblare LC-3 codul sursă al funcţiei main() de mai jos.

```
main()
{
    int a[5], i;
    i=4;
    while(i>=0){
            a[i]=i;
            i--;
    }
}
```

9. În secvenţa de cod următoare funcţia main() apelează funcţia *triple()*. Care este necesarul minim de memorie pentru zona de activare aferentă funcţiei *triple* ? Dar pentru zona de activare aferentă funcţiei *main* ?

```
main()
{
    int array[3];

    array[0]=1;
    array[1]=2;
    array[2]=3;
    triple(array);
}
```

BIBLIOGRAFIE

[Ack89] **Ackoff, R. L.**, *From Data to Wisdom*, Journal of Applies Systems Analysis, Volume 16, 1989 p 3-9.

[Aho86] **Aho A., Sethi R., Ullman J.** – *Compilers, Principles,Techniques and Tools*, Adison Wesley Publishing Comp. 1986.

[Bel04] **Bellinger G., Castro D., Mills A**., *Data, Information, Knowledge, and Wisdom*, 2004, http://www.systems-thinking.org/dikw/dikw.htm.

[Bud99] **Budiu M.**, *Teoria complexității*, http://www.cs.cmu.edu/~mihaib/

[Chi04] **Chitoşcă M.**, *Internetul ca agent de socializare al „Generației M*”, http://www.ris.uvt.ro/numarul52006/marituschitosca.pdf.

[Clc04] *Circuite logice combinaționale*, 2004, http://cid2004.3x.ro/

[Cor90] **Cormen, T. H., Leiserson, C. E., Rivest, R. L.** – *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[Fil97] **Filipescu V., Garaiman D.,** *Circuite electronice digitale. Îndrumar de laborator*, Reprografia Universității din Craiova, 1997.

[Flo03] **Florea A, Vinţan L.** – *Simularea şi optimizarea arhitecturilor de calcul în aplicaţii practice*, Editura MatrixROM, Bucureşti, 2003.

[Flo05] **Florea A.** – *Predicția dinamică a valorilor în microprocesoarele generației următoare*, Editura MatrixROM, Bucureşti, 2005.

[Gol97] **Golomety A.** – *Proiectarea Translatoarelor*, Editura Universității "Lucian Blaga", Sibiu, 1997.

[Gor05] **Gorgan D., Sebestyen-Pal Gh.**, *Computer Design*, Editura albastra, Cluj-Napoca,. ISBN 973-650-123-X, 2005.

[Katz05] **Katz R. H., Borriello G.,** *Contemporary Logic Design (2nd Edition)*, Prentice Hall, 2005.

[LC-3] *Little Computer v.3*. – www.ece.utexas.edu/~ambler/ee306/Software&Doc/**LC3**WinGuide.pdf.

[Lica06] **Lica D. ş.a.** – *Fundamentele programării*, Editura L&S Soft, Bucureşti, 2006.

[Log06] **Logofătu D.** – *Bazele programării în C. Aplicaţii*, Editura Polirom, Iaşi, 2006.

[Lun05] **Lungu V.**, *Procesoare INTEL, Programare în limbaj de asamblare*, Ediția a II-a, Editura Teora, Bucureşti, 2005.

[Mâr96] **Mârşanu Radu** – *Sisteme de calcul*, Manual pentru licee de informatică, cls. IX-a, Editura Didactică şi Pedagogică, Bucureşti, 1996.

[Min82] **Minsky M.**, *Why People Think Computers Can't*, AI Magazine, vol. 3, no. 4, Fall 1982, http://web.media.mit.edu/~minsky/papers/ComputersCantThink.txt.

[Mus97] **Muscă Gh.,** *Programare în limbaj de asamblare*, Editura Teora, Bucureşti, 1997.

[Neg97] **Negrescu L.** – „*Limbajul C. Manual pentru clasa a XI-a*”, Editura Computer Libris Agora, 1997.

[Nic04] **Niculae C.M.**, *Componente fundamentale*, 2004, http://fpce9.fizica.unibuc.ro/telecom/componente_fundamentale.htm

[Pal07] **Palsetia D.**, *Digital Systems Organization and Design*, 2007, http://www.cis.upenn.edu/~palsetia/cit595s07/

[Pat05] **Patterson D., Hennessy J.** – *Computer Organisation and Design*: *The Hardware/Software Interface*, Third Edition, Elsevier, 2005, ISBN: 1-55860-604.

[Patt03] **Patt Y., Patel S.** – *Introduction to Computing Systems: from bits & gates to C & beyond*, McGraw-Hill Higher Education, 2nd edition, 2003.

[Pop86] **Pop V.** – *Analiza şi sinteza dispozitivelor numerice*. Curs litografiat, Institutul Politehnic Timişoara, 1986, vol. I şi II.

[Sch95] **Schildt H.** – *C++ – manual complet*, McGraw-Hill Higher Education, 1995.

[Seb] **Sebestyen-Pal Gh.**, *Arhitectura calculatoarelor*, http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/

[Sha48] **Shannon C.E.**, *A Mathematical Theory of Communication*, Bell System Technical Journal, vol. 27, pp. 379-423, 623-656, July, October, 1948.

[Sto98] **Stoilescu D.** – „*Culegere de C / C++. Fişe de probleme*", Editura Radial, Galaţi, 1998.

[Vin00] **Vinţan N. L., Florea A.** – *Microarhitecturi de procesare a informaţiei*, Editura Tehnică, Bucureşti, ISBN 973-31-1551-7, 2000.

[Vin03] **Vinţan L.** – *Organizarea şi proiectarea microarhitecturilor*, 2003, http://webspace.ulbsibiu.ro/lucian.vintan/html/Organizarea.pdf.

[Wik] http://en.wikipedia.org/wiki/

[Zah04] **Zaharia M.H., Leon F.** – *Limbajul C de la Zero la Student*, Editura Politehnium Iasi, 2004.