

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

6.1. LC-3 – ARHITECTURA SETULUI DE INSTRUȚIUNI

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 ȘI SETUL DE REGISTRII GENERALI

Spațiul de memorie aferent unui sistem de calcul este organizat în așa fel încât să rețină **instrucțiunile** programelor utilizator, instrucțiunile programelor supervizor (aferente sistemului de operare) dar și **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 dispoziție doi regiștrii de interfață procesor – memorie:

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

LC-3 reprezintă un procesor virtual pe 16 biți (fiecare instrucțiune este codificată pe 16 biți). De asemenea, memoria este adresată printr-un cuvânt⁵ de 16 biți (busul de date fiind de 16 biți). În consecință, spațiul de memorie adresabil la LC-3 este de 2^{16} locații = 65536 (64k) locații. Adresabilitatea la LC-3 este tot de 16 biți, de unde rezultă faptul că, LC-3 dispune de o memorie totală de $64k \times 2o = 128ko$. Spre deosebire de alte procesoare, LC-3 nu este adresabilă pe octet. Spațiul de memorie al LC-3 (pentru detalii a se vedea și figura 10.4 din capitolul 10) conține o zonă de cod utilizator (începând cu adresa 0x3000), zona de date globale, zona rezervată sistemului de operare (care include și instrucțiunile rutinelor de serviciu), zona de stivă (aferentă apelurilor de funcții din programele utilizator – variabile locale, parametrii, etc.).

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

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

⁵ Cu toate că, la toate procesoarele octetul este format din 8 biți, numărul de biți care compun un cuvânt de date nu este identic la toate arhitecturile. Astfel, la LC-3 cuvântul (*word*) este pe 16 biți iar la procesorul MIPS R3000 este pe 32 de biți, informația stocată pe 16 biți 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 regiștrii generali $R_0 \div R_7$, fiecare pe 16 biți și trei regiștrii booleani 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 și 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 și apel de / revenire din subrutină valoarea PC-ului poate fi alterată.

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

Se reamintește că unitatea de control a procesorului LC-3 [Patt03] dispune de 2 regiștrii: PC și IR (registru care reține instrucțiunea aflată în curs de procesare). Instrucțiunile procesorului LC-3 sunt codificate pe un singur cuvânt⁶ de 16 biți și 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 și un set optimizat de instrucțiuni la un cost mai mic (vezi clasificarea procesoarelor RISC – CISC [Vin03] care face obiectul cursului de *Organizarea și proiectarea microarhitecturilor*). Acest compromis se stabilește 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:
 - **Regiștrii**: câte 3 biți pentru fiecare operand – sursă și / 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 față de 2** pe 16 biți. Celelalte procesoare (de exemplu MIPS, INTEL) implementează și tipurile: întreg pe 8 biți cu și fără semn (*short / byte, byte / unsigned byte*), întreg pe 16 biți cu și fără semn (*int / unsigned int, halfword / unsigned halfword*), întreg pe 32 biți cu și fără semn (*long / unsigned long, word / unsigned word*), simplă precizie virgulă mobilă – pe 32 de biți (*float*) respectiv dublă precizie virgulă mobilă – pe 64 de biți (*double*).

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

- **Imediat** (dacă operandul este localizat direct în instrucțiune).
- **Registru** (operandii sursă sunt doi regiștrii generali).
- **Memorie** (cu:
 - **Adresare directă** (sau relativă la PC): EA este codificată în corpul instrucțiunii și se obține printr-o însumare a PC cu câmpul generator de adresă pe 9 biți).
 - **Adresare indirectă**. În corpul instrucțiunii este codificat un pointer⁷ spre EA: valoarea PC + câmp generator de adresă pe 9 biți formează adresa la care găsim adresa operandului (și nu operandul ca în cazul adresării directe).
 - **Adresare indexată** (sau relativă la o adresă de bază) – folosită în cazul instrucțiunilor de prelucrare ale tablourilor (structurilor matriceale). EA se obține însumând adresa de bază stocată într-un registru general și un câmp generator de adresă pe 6 biți. Diferența dintre adresarea directă și cea indexată constă în faptul că, în cazul primeia, adresarea se face într-o zonă (NEAR) – apropiată PC-ului instrucțiunii de transfer date (+/- 256 instrucțiuni) iar în cazul celei de-a doua adresări, spațiul accesat poate fi (FAR) îndepărtat față de instrucțiunea de transfer (practic aproape oriunde în zona de cod utilizator).

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

- ❖ **Instrucțiuni operaționale**, care manipulează direct date. Din această categorie fac parte cele aritmetico-logice (ADD, AND și NOT).
- ❖ **Instrucțiuni de transfer date**:
 - a) între memorie și regiștrii procesorului
 - citire din memorie (LD, LDI și LDR)

⁷ Termenul de *pointer* a fost preluat în limba română și poate fi folosit cu sensul de referință, indicator de adresă, localizator. Pentru detalii studiați capitolul 12.

- scriere în memorie (ST, STI și 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 / registre și *porturi* (registre de interfață ai dispozitivelor periferice – registre de stare sau de date ai tastaturii sau ai monitorului – vezi pentru detalii capitolul 8 *Înteruperi software*).

Instrucțiuni de control (salt condiționat / necondiționat, direct / indirect și 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 și NOT. Operanzii destinație ai acestora sunt registrele generali ai procesorului dar semnul acestuia va seta și unul din cei trei registre 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 și 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 și una de scriere înapoi în memorie.

Instrucțiunile ADD și AND pot opera și în modul imediat de adresare. Pentru fiecare din instrucțiunile setului LC-3 ISA sunt ilustrate grafic codificarea și 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 și 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 și [Patt03]) „*completitudinea porților ȘI-NU*” – adică prin intermediul acestor porți pot fi implementate oricare operații logice – LC-3 ISA oferă doar instrucțiunile NOT și AND, celelalte instrucțiuni (OR, XOR) existente la alte arhitecturi (MIPS, INTEL) putând

fi realizate prin combinații ale instrucțiunilor NOT și AND (vezi spre rezolvare problemele 9 și 10 de la sfârșitul acestui capitol).

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

Instrucțiunea NOT

NOT Reg[Dst], Reg[Src]

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

Semantica instrucțiunii ($\text{Reg}[\text{Dst}] \leftarrow \text{NOT } \text{Reg}[\text{Src}]$) presupune inversarea fiecărui bit al registrului sursă (transformarea în complement față de 1) și copierea pe poziția corespunzătoare în registrul destinație. Registrul sursă rămâne nemodificat în urma execuției instrucțiunii. De menționat că sursa și destinația pot referi același registru.

Exemplu:

NOT R2, R6



Setul de regiștrii generali

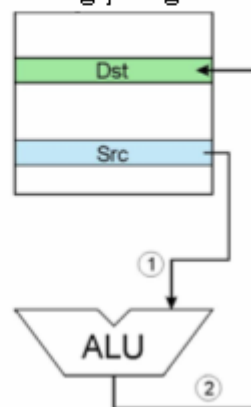


Figura 6.1. Codificarea instrucțiunii NOT și succesiunea de operații efectuate în cazul acesteia

La începutul fazei de *decodificare* câmpul IR[8:6] va ataca setul de regiștrii 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. Declanșarea acestuia (NOT) se realizează în faza de *execuție* urmată apoi de scrierea rezultatului în setul de regiștrii generali (pe timpul fazei *scriere rezultat*).

Instrucțiunile ADD/AND

ADD Reg[Dst], Reg[Src1], Reg[Src2] sau **ADD** Reg[Dst], Reg[Src1], Imm5
și
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 regiștrii, fie un registru și o valoare imediată). Registrul destinație este identificat prin câmpul IR[11:9] iar unul dintre regiștrii 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 pe 16 biți obținându-se un număr întreg în complement față de 2. Se reamintește că, operațiile aritmetico-logice se efectuează cu operanzi de dimensiuni egale în număr de biți. Semantica celor două instrucțiuni este următoarea:

$\text{Reg}[\text{Dst}] \leftarrow^8 \text{Reg}[\text{Src1}] + \text{SEXT}(\text{Imm5})$ – în cazul adunării și respectiv

$\text{Reg}[\text{Dst}] \leftarrow \text{Reg}[\text{Src1}] \text{ AND } \text{SEXT}(\text{Imm5})$ – în cazul operației de ȘI 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**, deși 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 șirul „0x” care sugerează o valoare în sistemul hexazecimal de numerație.

⁸ Simbolul \leftarrow are rolul de atribuire a valorii expresiei din dreapta sa variabilei din stânga sa.

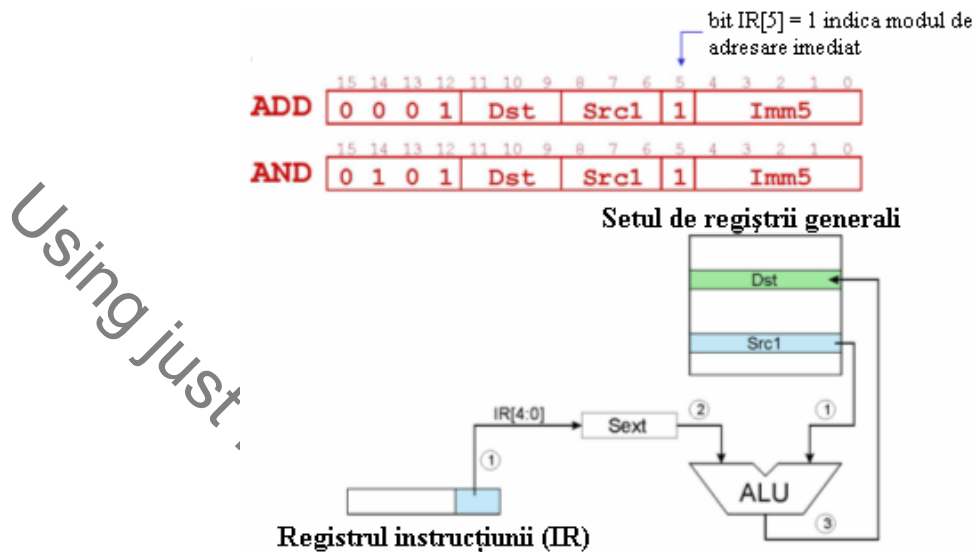


Figura 6.2. Codificarea instrucțiunilor ADD/AND cu mod de *adresare imediat* și succesiunea de operații efectuate în cazul acestora

Dacă modul de adresare este registru atunci bitul $IR[5] = 0$, biții $IR[4:3]$ sunt și ei 0 iar câmpul $IR[2:0]$ specifică cel de-al doilea registru sursă. Semantica celor două instrucțiuni este următoarea:

$Reg[Dst] \leftarrow Reg[Src1] + Reg[Src2]$ – în cazul adunării și respectiv

$Reg[Dst] \leftarrow Reg[Src1] \text{ AND } Reg[Src2]$ – în cazul operației de ȘI logic

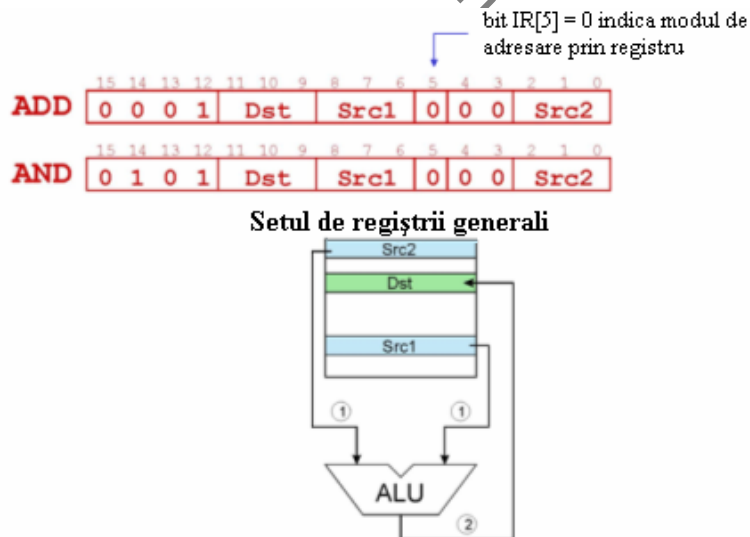


Figura 6.3. Codificarea instrucțiunilor ADD/AND cu mod de *adresare prin registru* și succesiunea de operații efectuate în cazul acestora

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 și 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 registre generale nu permite cablarea unui registru (R_0) la masă, cum fac alte procesoare RISC [Flo03].

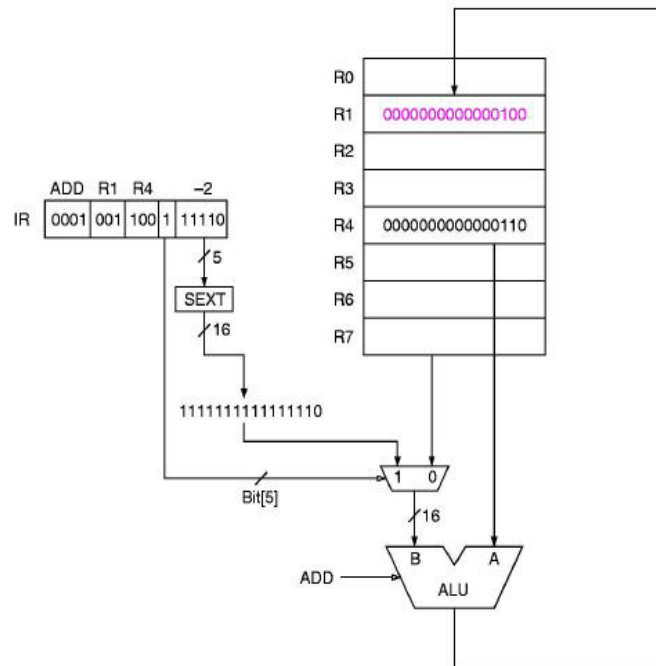


Figura 6.4 Exemplu de adunare cu mod de adresare imediat

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

Întrucât LC-3 pune la dispoziție 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*) și LDR (*load indexat*) și respectiv, trei instrucțiuni distincte pentru scrierea datelor din regiștrii procesorului în memorie: ST (*store direct*), STI (*store indirect*) și STR (*store indexat*).

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

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

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

Problema principală în cazul instrucțiunilor cu referire la memorie o reprezintă modul de adresare, care intenționează specificarea adresei direct în instrucțiune. Dar cum spațiul de adresare este de 2^{16} locații și instrucțiunile sunt pe 16 biți, mai mult, 4 biți stabilesc codul operației iar alți trei biți registrul sursă / destinație, rezultă că doar maxim 9 biți din instrucțiune pot fi utilizați la calculul adresei. Drept soluție, se poate considera memoria LC-3 ca și o colecție 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 biți (sau 6 la adresarea indexată) constituie deplasamentul (*offset-ul*) în respectiva pagină.

Instrucțiunile LD și ST

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

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

$$EA \leq (PC) + \text{SEXT}(\text{IR}[8:0])$$

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

Semantica celor două instrucțiuni este următoarea:

LD Reg[Dst], *Label* $\text{Reg}[\text{Dst}] \leftarrow \text{Mem}[(\text{PC}) + \text{SEXT}(\text{IR}[8:0])]$
 ST Reg[Src], *Label* $\text{Reg}[\text{Src}] \Rightarrow \text{Mem}[(\text{PC}) + \text{SEXT}(\text{IR}[8:0])]$

În cele două mnemonici asamblare LD Reg[Dst], *Label* și 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 și LEA) codurile de condiție (N, Z și P) sunt setate corespunzător.

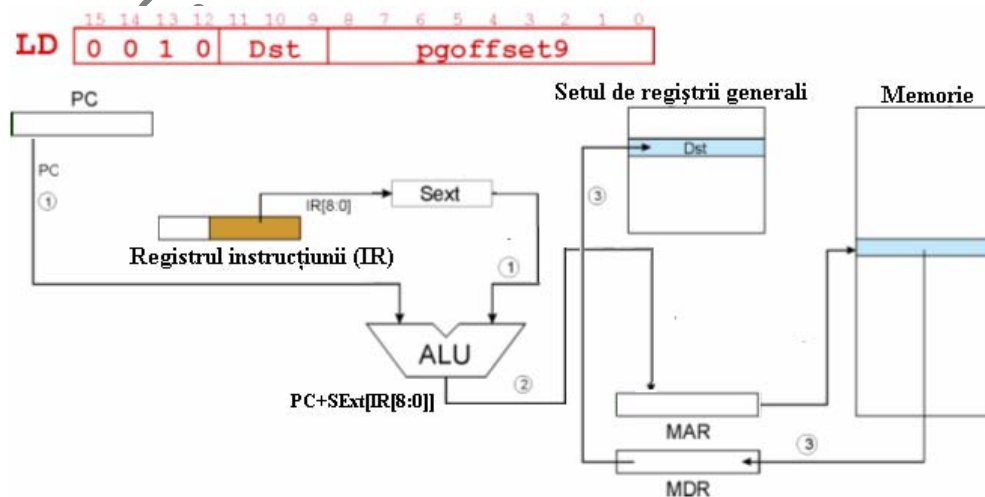


Figura 6.5. Codificarea instrucțiunii Load cu mod de adresare direct și succesiunea de operații efectuate în cazul acestora

În figura 6.5 sunt prezentate codificarea instrucțiunii LD și succesiunea operațiilor ce concură la execuția acesteia. Astfel, pe timpul fazei de *decodificare*, PC-ul instrucțiunii LD și 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 rezultând 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 registre generale la locația specificată de câmpul Dst (IR[11:9]) (etapa 3).

Exemplu:

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

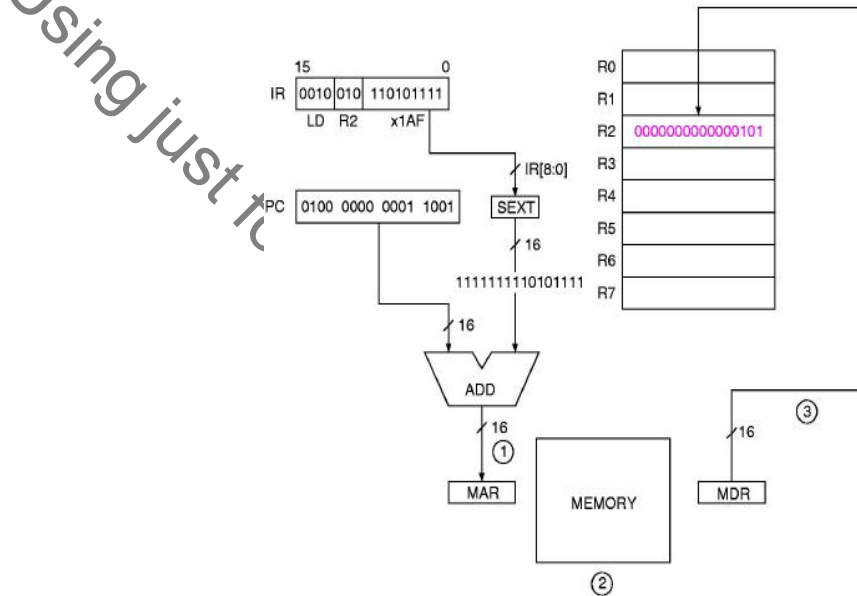


Figura 6.6. Exemplu de instrucțiune Load cu mod de adresare direct

Pentru operația de scriere în memorie (instrucțiunea **Store** cu mod de adresare direct) diferența în ce privește codificarea este următoarea:

0011 Src poffset9

În urma execuției instrucțiunilor de scriere în memorie (ST, STI și STR) codurile de condiție rămân nemodificate. În ce privește modul de operare a instrucțiunii Store cu mod de *adresare direct*, succesiunea de operații efectuate este aceeași cu cea din figura 6.5 cu deosebirea că operația 3 se face în sens invers (dinspre setul de regiștrii generali, de la un registru sursă de această dată, spre o locație de memorie, prin intermediul aceluiași registru MDR (registru de date a memoriei).

Instrucțiunile LDI și STI

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

O altă modalitate de a obține o adresă completă (pe 16 biți) prin intermediul unei instrucțiuni o reprezintă citirea adresei dintr-o locație 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).

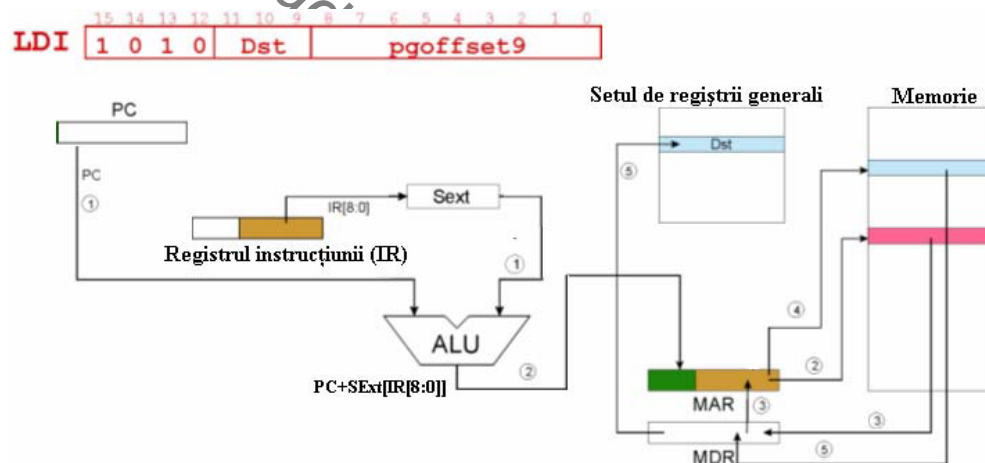


Figura 6.7. Codificarea instrucțiunii Load cu mod de adresare indirect și succesiunea de operații efectuate în cazul acestora

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,

$$pointer\ address \leq (PC) + SEXT(IR[8:0])$$

$$EA\ (adresa\ efectivă) \leq Mem\ [pointer\ address]$$

$$Reg\{Dst\} \leq Mem\ [Mem\ [pointer\ address]]$$

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

Modul de operare al instrucțiunii LDI este următorul: pe timpul fazei de *decodificare* se atacă sumatorul (unitatea ALU) cu PC-ul instrucțiunii LDI și câmpul generator de adresă IR[8:0] căruia i se extinde semnul pe 16 biți (etapa 1 din figura 6.7). În faza de *evaluare adresă* se obține 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 și conținutul locației 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 *execuție* se accesează memoria din nou cu adresa tocmai stocată în MAR (etapa 4 din figura 6.7) și se obține operandul care se depune în registrul MDR (etapa 5 din figura 6.7). În final, pe parcursul fazei *Scrie rezultat*, conținutul lui MDR este depus în setul de regiștrii generali la locația specificată de câmpul Dst (IR[11:9]) (etapa 5).

Exemplu:

Se consideră la adresa PC = 0x4A1C instrucțiunea **LDI R3, 0x1CC**, iar în memorie la locația 0x49E8 se află valoarea 0x2110. De asemenea, la locația 0x2110 se găsește valoarea 0xFFEF. Figura 6.8 ilustrează modul de lucru al instrucțiunii LDI cu stocarea rezultatului final în registrul destinație R3.

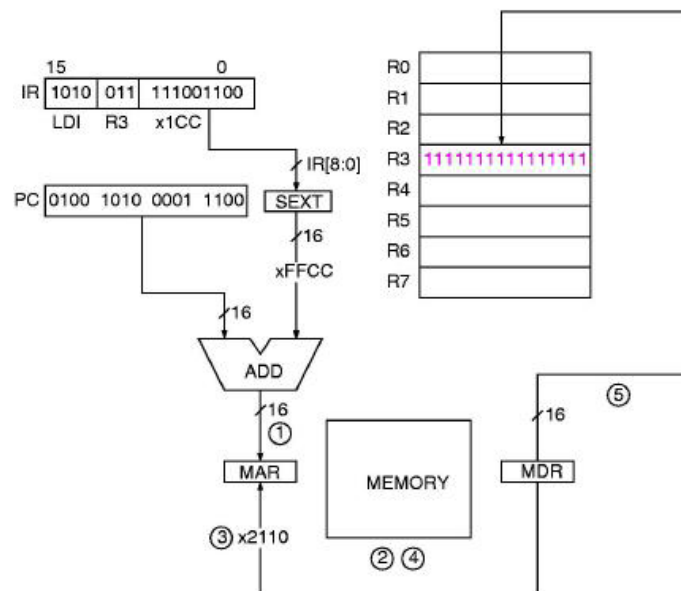


Figura 6.8. Exemplu de instrucțiune Load cu mod de adresare indirect

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

1011 Src poffset9

În ce privește modul de operare a instrucțiunii Store cu mod de adresare indirect, succesiunea de operații efectuate este aceeași 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 regiștri generali, de la un registru sursă de această dată, spre o locație de memorie, prin intermediul aceluiași registru MDR (registru de date a memoriei)).

Instrucțiunile LDR și STR

LDR Reg[Dst], Reg[BaseR], index6 și STR Reg[Src], Reg[BaseR], index6

În modul de adresare direct pot fi adresate cuvinte aflate în aceeași pagină de memorie cu instrucțiunea curentă. Totuși, în cazul programelor mari, acest lucru nu este suficient spațiul fiind prea mic, chiar și 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 folosește 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 și 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,

$$EA \leq \text{Reg}[\text{BaseR}] + \text{ZEXT}(\text{IR}[5:0])$$

$$\text{Reg}[\text{Dst}] \leq \text{Mem}[\text{Reg}[\text{BaseR}] + \text{ZEXT}(\text{IR}[5:0])]$$

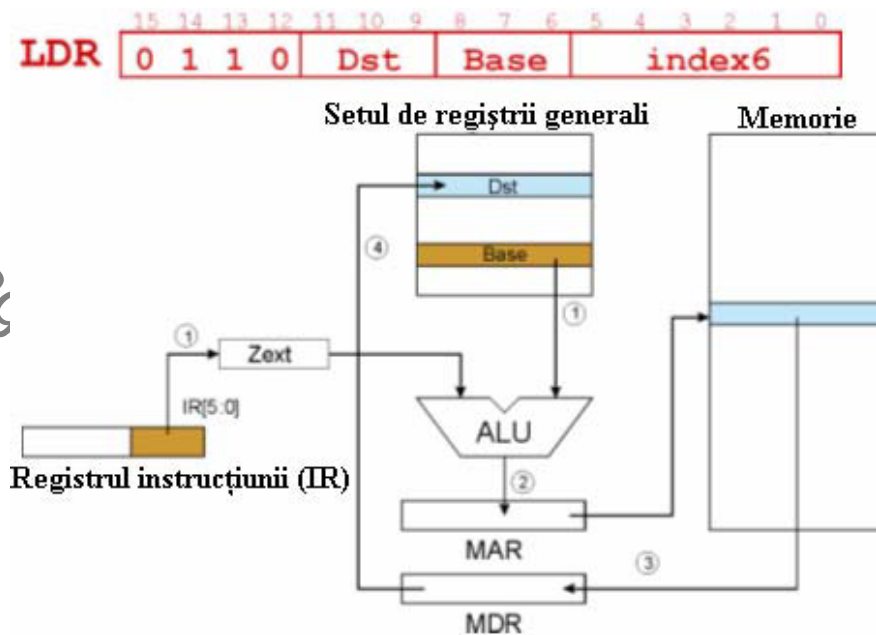


Figura 6.9. Codificarea instrucțiunii Load cu mod de *adresare indexat* și succesiunea de operații efectuate în cazul acestora

Ciclul instrucțiunii LDR (*load indexat*) presupune după aducerea și depunerea acesteia în registrul IR (lucru efectuat în faza *fetch instrucțiune*) *decodificarea* instrucțiunii. Pe parcursul acestei faze, cu cei trei biți IR[8:6] se identifică registrul de bază din setul de registre generali și se realizează extensia cu 0 pe 16 biți a câmpului deplasament IR[5:0] (etapa 1 din figura 6.9). Pe timpul fazei de *evaluare adresă* se calculează suma dintre cele două informații 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 conținutul se depune în registrul MDR (etapa 3 din figura 6.9), de unde urmează a fi copiat în registrul destinație corespunzător în faza *Scrie rezultat* (etapa 4 din figura 6.9).

Exemplu:

Se consideră instrucțiunea **LDR R3, R3, 0x1D**, iar în memorie la locația 0x2362 se află valoarea 0x0F0F. De asemenea, conținutul registrului R3 înaintea execuției instrucțiunii este 0x2345. Figura 6.10 ilustrează modul de lucru al instrucțiunii LDR cu stocarea rezultatului final în registrul destinație (același) R3.

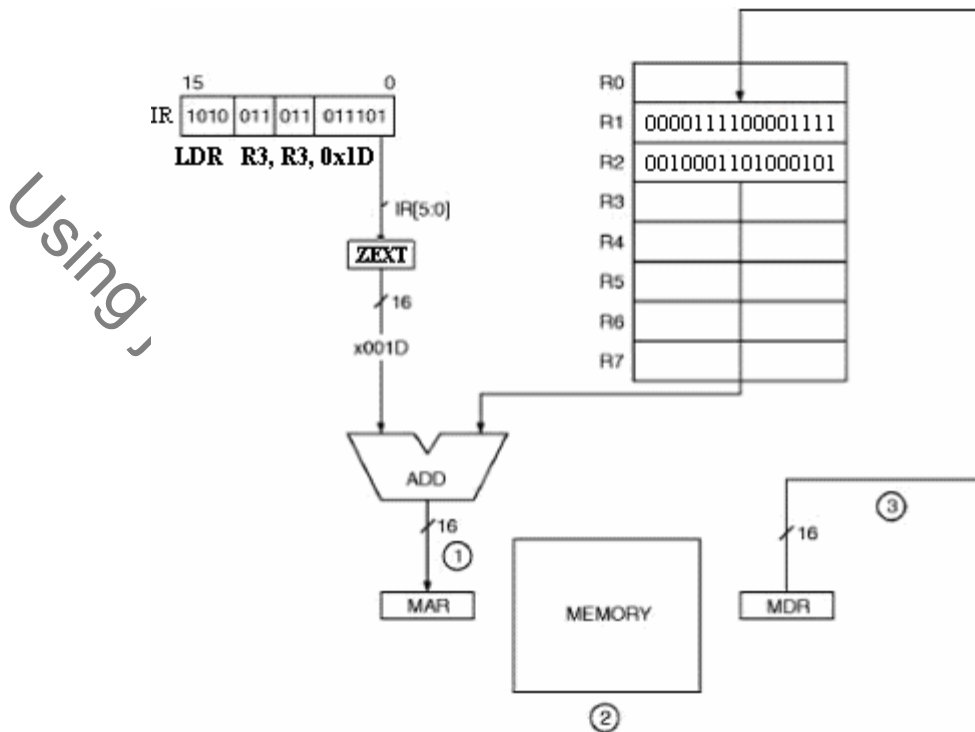


Figura 6.10. Exemplu de instrucțiune Load cu mod de adresare indexat

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,

$$EA \leq (PC) + \text{SEXT}(\text{IR}[8:0])$$

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

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

$$\text{LEA Reg[Dst], Label} \quad \text{Reg[Dst]} \leq (PC) + \text{SEXT}(\text{IR}[8:0])$$

Uzual se folosește 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 și 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.

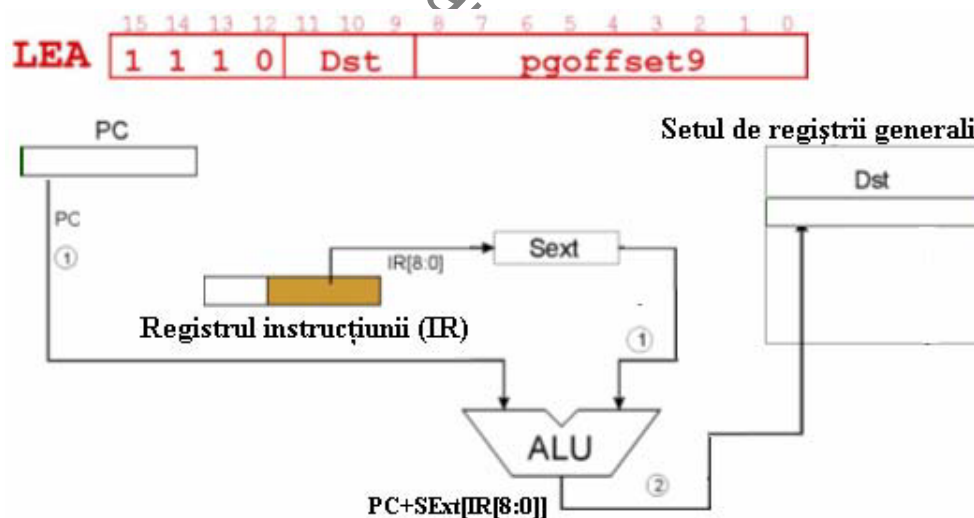


Figura 6.11. Codificarea instrucțiunii Load cu mod de adresare imediat și succesiunea de operații efectuate în cazul acestora

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 (același) R5.

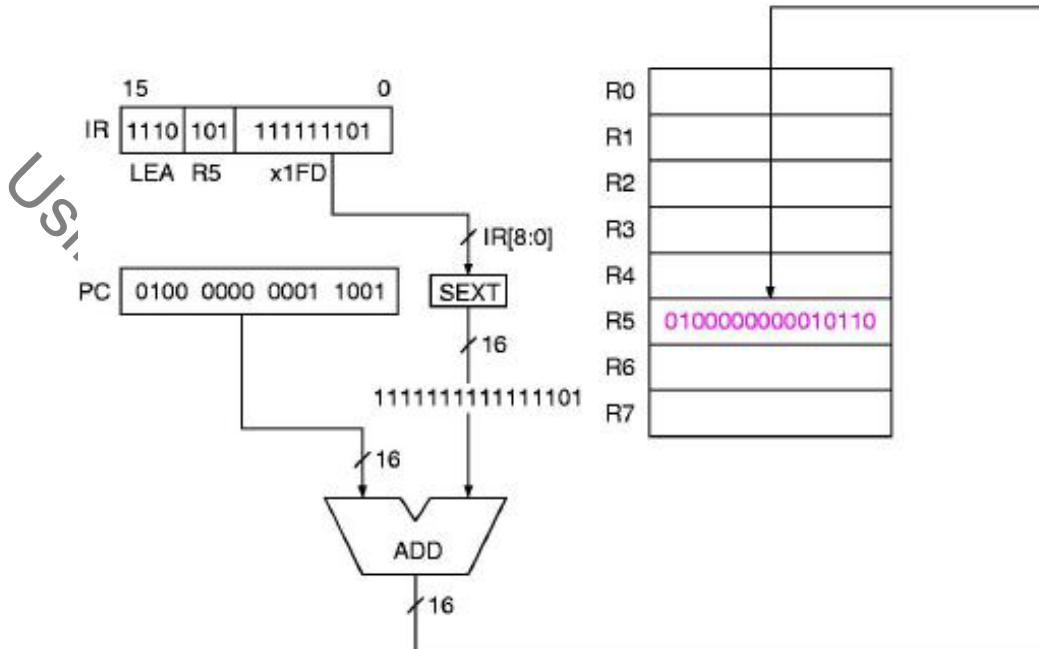


Figura 6.12. Exemplu de instrucțiune Load cu mod de adresare imediat

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

Instrucțiunile de ramificație (de salt și / 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 îndeplinește condiția) sau **necondiționate** (se face tot timpul saltul – PC-ul se modifică întotdeauna). În cazul saltului condiționat dacă nu se îndeplinește 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 cunoaște 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-și modifică în mod dinamic valoarea reținută [Flo05]). În cazul salturilor indirecte, adresa de salt se cunoaște abia în momentul execuției programului.

Pe lângă instrucțiunile de salt mai există și instrucțiuni de apel și 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 totuși chiar și 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 și 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 (*pgoffset* din figura 6.13) căruia i se extinde semnul de la 9 la 16 biți. Astfel,

$$EA \leq (PC) + \text{SEXT}(\text{IR}[8:0])$$

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

Simbolul *x* reprezintă condiția care trebuie îndeplinită și 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 (registrii booleani N, Z și P) sunt 0 (nu sunt setați practic), conform figurii

6.14 niciuna din cele trei porți logice ȘI nu generează 1 logic (toate ieșirile sunt 0) și rezultă că saltul nu se face (echivalent unui simplu NOP – *no operation*) și se trece la următoarea instrucțiune din program succesoare branch-ului. Dacă registrul boolean corespunzător condiției este setat (de ex. condiția „n” – negativ și 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 și 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 regiștrii booleani setați (de ex. condiția „n” – negativ și 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 și figura 6.13).

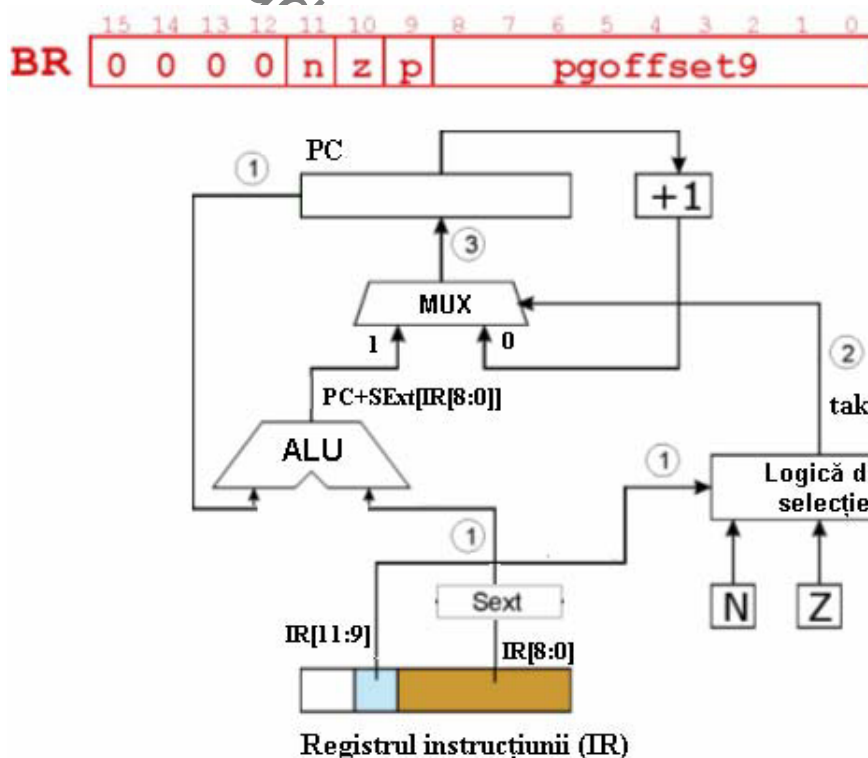


Figura 6.13. Codificarea instrucțiunii *Branch* și succesiunea de operații efectuate în cazul acesteia

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

registru IR are loc și incrementarea PC-ului cu o unitate pentru a indica spre următoarea instrucțiune din program (valoarea PC+1 – unde PC reprezintă adresa branch-ului – fiind depusă pe intrarea 0 a multiplexorului MUX și 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 biți și PC-ul saltului se depun pe intrarea sumatorului ALU (etapa 1 din figura 6.13) iar câmpul de condiție IR[11:9] atacă blocul „logică de selecție” – prezentat detaliat în figura 6.14 (practic 3 porți logice ȘI cu două intrări și o poartă logică SAU cu 3 intrări), a cărui ieșire va fi semnalul *taken* (îndeplinirea condiției 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 ($PC + \text{SEXT}(\text{IR}[8:0])$) care va ataca a doua intrare a multiplexorului (cea de 1 logic). În faza de *execuție* în funcție de îndeplinirea sau nu a condiției de salt în PC se încarcă valoarea corectă, adresa instrucțiunii de la care se va procesa în continuare (etapa 3 din figura 6.13).

Spre exemplificare, se consideră instrucțiunea de salt **BR z 0x0D9** stocată în memorie la adresa PC=0x4028. De asemenea, dintre regiștrii booleani de condiție doar registrul Z este setat. Figura 6.14 ilustrează succesiunea de operații care conduc la modificarea fluxului de control al programului și determinarea noului PC

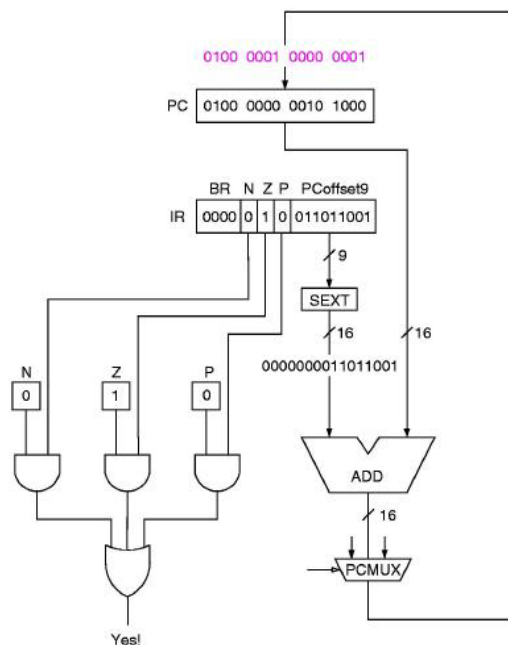


Figura 6.14. Exemplu de instrucțiune de salt condiționat (*Branch if zero*)

Tratarea instrucțiunilor de salt necondiționat, apel și revenire din subrutine utilizator și ale sistemului de operare, precum și 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 și JSR respectiv JMPR / JSRR este aproape identic, un singur bit diferențiind între salt și respectiv apel (păstrarea PC-ului de revenire în registrul R7 la LC-3 [Patt03]). Asupra formatului celor patru instrucțiuni (JMP/JSR și 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, afișarea unui caracter pe ecranul monitorului sau încheierea programului utilizator și 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 programe⁹. La nivelul programelor în limbaj de asamblare implementarea buclilor 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 pași*) [Neg97, Sto98], fie prin utilizarea unei santinelle¹⁰ 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 pași*).

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

¹⁰ Un caracter special folosit pentru a indica sfârșitul unei secvențe este numit deseori **santinellă**. Utilitatea sa este foarte mare atunci când nu se cunoaște apriori numărul de pași (iterații) pe care îi execută o anumită structură repetitivă.

Aplicație 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 regiștrilor R1, R2, R3 și 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.

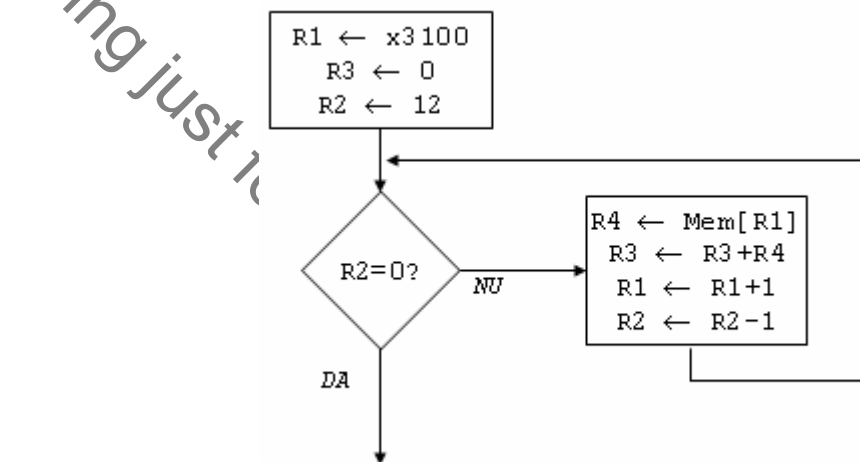


Figura 6.15. Schema logică aferentă programului de însumare a 12 numere întregi

Secvența de cod aferentă este următoarea:

PC	Codificare instrucțiuni	Semnificație
0x3000	1110 001 1 0000 0000	$R1 \leftarrow x3100$
0x3001	0101 011 011 1 00000	$R3 \leftarrow 0$
0x3002	0101 010 010 1 00000	$R2 \leftarrow 0$
0x3003	0001 010 010 1 01100	$R2 \leftarrow 12$
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. „Fișierul” 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 fișier. Adresa de început a fișierului 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 fișier (cu cel citit la momentul respectiv din fișier) se incrementează un contor. Sfârșitul de fișier este indicat prin codul ASCII special **EOT (x04)** – “end of text”. În final, programul va afișa numărul de caractere din fișier 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.

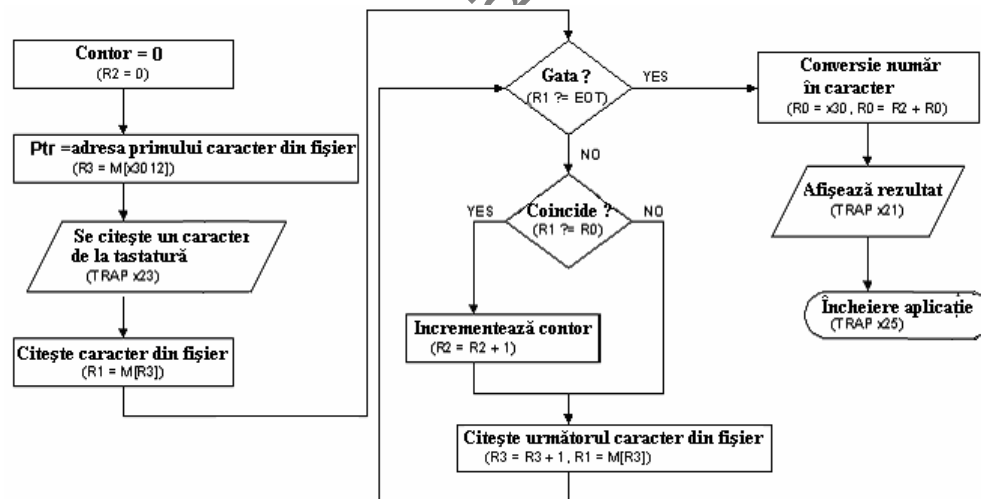


Figura 6.16. Schema logică aferentă programului care determină numărul de apariții ale unui caracter într-un fișier

În secvența de program următoare Contorul (numărul de apariții ale caracterului căutat) este salvat în registrul R2, registrul R3 păstrează adresa de început a fișierului (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 fișier.

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 fișier.

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

0x3012	Adresa de început a fișierului	
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*) și este susținută printr-o serie de componente *hardware*. Strategia fluxului de date presupune luarea fiecărui element component, determinarea modului de funcționare, identificarea intrărilor / ieșirilor, 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 și pe 1, 2, 3, 9 sau 11 biți) și „**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 îngroșată, cu săgeți *pline* la ambele capete. Magistrala globală la LC-3 constă din 16 fire (semnale) și 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 (și nu de mai multe). Structura care dorește 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. Regiștrii capturează data de pe bus doar dacă *unitatea de control* activează semnalul Write Enable pentru destinația corespunzătoare.



S

S

– 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 și „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 și 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 și 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* și evident a fluxului datelor la LC-3 ISA, acceptă ca și intrări fie doi regiștrii sursă din **setul de regiștrii generali**, selectați cu câmpurile IR[8:6] și respectiv IR[2:0] (SR1 și SR2 în figura 6.17), fie un registru sursă și 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 și 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 regiștrii generali la locația specificată prin registrul destinație corespunzător instrucțiunii procesate. De asemenea, rezultatul va fi folosit de către structura „LOGIC” pentru a seta regiștrii booleani de condiție (N, Z și P). Pe lângă instrucțiunile aritmetico-logice (ADD, AND și NOT) și instrucțiunile de transfer date în regiștrii (LD, LDI, LDR și LEA) setează codurile de condiție prin intermediul bus-ului global.

Setul de regiștrii generali, componentă esențială a fluxului datelor, este o structură *biport* pentru citire și *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 și câmpul IR[8:0] căruia i se extinde semnul de la 9 la 16 biți. O altă intrare în PCMUX va fi obținută din setul de registre 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). Deși 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 își 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 mașină, 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 registre vor fi scriși (WE va fi 1 pentru care structură logică), ce operație va efectua unitatea aritmetico-logică ?

6.3. EXERCITII ȘI PROBLEME

1. Se consideră cunoscute valorile inițiale pentru registre și 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 și codificarea fiecăreia dintre ele, specificând și 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 și 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 $(R1 \times R1) + 6$, punând rezultatul în R3. Se cere pentru început să translați programul din cod mașină (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 “ $R0 \leftarrow \text{NOT } R1$ ” 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 și R3 rețin numere întregi în complement față de 2 pe 16 biți, pentru ce valori ale lui R1 rezultatul din R3 este corect. Justificați.

4. a) Instrucțiunile LC-3 următoare (A și B) realizează în principiu același lucru. Explicați în cel mult 10 cuvinte ce fac ele.

A 0000 0001 0101 0101

B 0001 0000 0010 0000

Totuși, ele nu pot fi interschimbate deoarece nu fac chiar același lucru. În cel mult 10 cuvinte explicați care este diferența între cele două instrucțiuni.

b) În cadrul LC-3 ISA sunt adunate următoarele două numere întregi în complement față de 2 pe 16 biți: 010101011010101 și 0011100111001111 obținându-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 dorește execuția unei instrucțiuni care să scadă valoarea întregă 20 din registrul R1 și să depună rezultatul în R2. Poate fi realizat acest lucru ? Dacă *da* scrieți mai jos codificarea instrucțiunii. Dacă *nu* explicați care este problema.

15																0
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	---

d) Ce specifică arhitectura setului de instrucțiuni aferentă unui procesor. Care sunt diferențele dintre instrucțiunile LD, LDI și LDR ? Dar diferența dintre acestea și instrucțiunea LEA ?

e) Dacă dimensiunea totală a memoriei este de 1MB (2^{20} octeți) și adresabilitatea este de 32 de octeți, câți biți de adresă sunt necesari pentru adresarea întregului spațiu de memorie ?

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

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

PC	Codificare instrucțiuni	Codificare de tip transfer registru (RTL)
x3001	0101 0100 1010 0000	$R2 \leftarrow 0$
x3002	0001 0100 1010 0001	$R2 \leftarrow R2 + 1$
x3003	0001 0100 1000 0010	$R3 \leftarrow R2 + R2$
x3004	0001 0100 1000 0011	$R2 \leftarrow R2 + R3$
x3005	0001 0010 0111 1111	$R1 \leftarrow R1 - 1$
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 și 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 și care cuprinde instrucțiunile cu semantica din ultima coloană a tabelului următor:

PC	Codificare instrucțiuni	Codificare de tip transfer registru (RTL)
x3001		$R2 \leftarrow M[R1+0]$
x3002		$R3 \leftarrow M[R1+1]$
x3003		$R4 \leftarrow \text{NOT } R3$
x3004		$R4 \leftarrow R4 + 1$
x3005		$R5 \leftarrow R2 + R4$
x3006		BRzp x3009

x3007		$M[R1+2] \leftarrow R3$
x3008		BRnzp x3010
x3009		$M[R1+2] \leftarrow R2$

a) Completați tabelul cu codificarea fiecărei instrucțiuni.

b) Parcurgeți programul pas cu pas începând cu adresa 0x3001 și 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. Următoarele colane redau starea inițială a regiștrilor generali și a celor de condiție din LC-3 ISA. Se presupune de asemenea că la adresa de memorie 0x3100 se găsește valoarea 14 iar la adresa 0x3101 se găsește valoarea 27.

PC	Operația	R0	R1	R2	R3	R4	R5	R6	R7	CCs
	Starea inițială a regiștrilor →	0	x3100	0	3	4	5	6	7	
x3001	$R2 \leftarrow M[R1+0]$			14						P
x3002										
x3003										
x3004										
x3005										
x3006										
x3007										
x3008										
x3009										

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

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

PC	Codificare instrucțiuni	Codificare de tip transfer registru (RTL)
	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
x3001	0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0	$R2 \leftarrow M[R1+0]$
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 și 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 regiștrilor generali și a celor de condiție din LC-3 ISA. Se presupune de asemenea că la adresa de memorie 0x3100 se găsește valoarea -34 iar la adresa 0x3101 se găsește valoarea -20.

PC	Operația	R0	R1	R2	R3	R4	R5	R6	R7	CCs
Starea inițială a regiștrilor →			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 regiștrii R1 și R2 iar rezultatul este depus în registrul R3. Completați cele două instrucțiuni care lipsesc pentru obținerea rezultatului dorit.

(1): 1001 100 001 11111

(2): _____

(3): 0101 110 100 000 101

(4): _____

10. a) Dispunând doar de instrucțiunile ADD, AND și NOT cum poate fi implementată o instrucțiune de scădere (descrieți succesiunea de instrucțiuni care concură la execuția unei operații de scădere):

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

- b) Aceeași problemă dar pentru instrucțiunea

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

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

MOVE Reg[Dst], Reg[Src1].

- d) Cum inițializăm un registru cu 0 ?

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

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

	<i>Fetch Instrucțiune</i>	Decodificare	Evaluare Adresă	Fetch Operand	Execuție	Scrie Rezultat
PC						
IR						
MAR						
MDR						

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

	<i>Fetch Instrucțiune</i>	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):

	<i>Fetch Instrucțiune</i>	Decodificare	Evaluare Adresă	Fetch Operand	Execuție	Scrie Rezultat
PC						
IR						
MAR						
MDR						

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

	<i>Fetch Instrucțiune</i>	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.

- ADD R1, R2, R3
- LD R1, LABEL
- 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 MAȘINĂ. 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 și C/C++, fiind aceea că el oferă doar câteva tipuri simple de comenzi și 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 mașină.

În ciuda unor dezavantaje clare, specifice programării în limbaj de asamblare: *lizibilitate greoaie a codului* (greu de scris, citit și înțeles), *depanare și 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 și 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 și în funcții ale sistemului de operare), este necesar un **timp redus de execuție** din partea lor și să ocupe un **cât mai puțin spațiu de memorie**. Programarea în limbaj de asamblare este în primul rând dependentă de mașina (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 și 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 și 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 și acumulatori) și **memorii de capacități relativ reduse**.

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

De asemenea, **medii integrate de programare-dezvoltare** și 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 și 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 și 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 urmărește 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 numește **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 mașină** (succesiune de 0 și 1 – extrem de prietenoasă procesorului și întregului sistem de calcul 0001110010000001). Procesul de asamblare

realizează o corespondență între fiecare simbol (element de limbaj asamblare) și setul de instrucțiuni specificat de ISA-ul mașinii 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 și inițializare date. Trebuie adăugat că unei instrucțiuni dintr-un limbaj de nivel înalt (ex: C, Pascal, etc.) îi corespunde o secvență de instrucțiuni mașină (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 fișiere de asamblare LC-3 încep cu simbolul ‘;’ (ca și la INTEL). Orice urmează acestui caracter până la sfârșitul liniei este ignorat. Identificatorii sunt o secvență de caractere alfanumerice, linie de subliniere, și trebuie să nu înceapă cu un număr. Opcode-ul instrucțiunilor sunt cuvinte rezervate care nu pot fi folosite ca identificatori. Etichetele sunt declarate prin așezarea lor la începutul unei linii. Numerele sunt implicit în baza 10 și sunt precedate de simbolul #. Dacă sunt precedate de caracterul 0x, ele sunt interpretate în sistemul de numerație hexazecimal. Deci, #256 și 0x100 semnifică aceeași valoare.

Șirurile sunt încadrate de ghilimele ‘”’. Caracterele speciale din șiruri 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 translate în instrucțiuni ale limbajului mașină (LM). Deși se aseamănă cu instrucțiunile (caracterizate de un cod de operație) ele sunt precedate întotdeauna de „.” (simbolul punct). Sunt utilizate de asamblor pentru a

aloca, inițializa zone de memorie de date necesare în program dar și pentru a marca începutul și sfârșitul 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 execuție a codului mașină.
.END		Marchează sfârșitul programului sursă. Atenționează asamblorul să nu mai incrementeze contorul de linii în etapa de generare a tabeli de simboluri.
.BLKW	n	Alocă <i>n</i> cuvinte de spațiu 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 și îl inițializează cu valoarea <i>val</i> .
.STRINGZ	n - character string	Alocă <i>n</i> +1 locații de memorie (cuvinte) și salvează în această zonă șirul de caractere respectiv (câte un cuvânt per locație). În locația <i>n</i> +1 este salvat caracterul x0000 – terminatorul NULL – care marchează sfârșitul șirului.

Tabelul 7.1. Directivele de asamblare ale LC-3

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

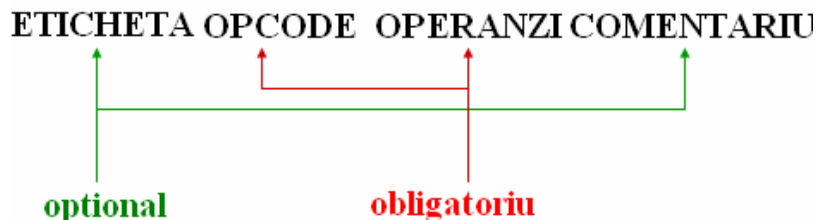


Figura 7.1. Formatul instrucțiunilor în limbajul de asamblare LC-3

Opcode – **ul** reprezintă cuvinte rezervate care nu trebuie să fie folosite pe post de identificatori, etichete și care corespund instrucțiunilor din setul LC-3 ISA (în limbaj mașină opcode-ul este reprezentat pe 4 biți). Toate instrucțiunile și implicit opcode-urile sunt ilustrate în subcapitolul 7.2.1 (ex: ADD, AND, LD, LDR, ...).

Operanzii pot exista sub următoarea formă:

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

Operanzii sunt separați prin virgulă. Numărul, ordinea și tipul corespund formatului instrucțiunii (vezi subcapitolul 7.2.1).

Ex:

ADD	R1, R1, R3	$R1 \leftarrow R1 + R3$
ADD	R1, R1, #3	$R1 \leftarrow R1 + 3$
LD	R6, NUMBER	$R6 \leftarrow \text{MEM}[\text{NUMBER}]$
BRz	LOOP	Execută salt la LOOP pe condiție egală cu zero.

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

Ex1

LOOP	ADD	R1, R1, #-1
	BRp	LOOP

Ex2:

.	.
.	LD R2, SIX
.	.
.	.
SIX	.
	.FILL 6

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

- Trebuie evitată comentarea unor lucruri evidente: de ex. “se decrementează R1”.
- Furnizați 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 și înțelegerii codului scris în limbaj de asamblare, dar nu numai.

1. Realizați un antet cu numele autorului, eventuale date de contact și scopul programului
2. Etichetele, opcode-ul, operanzii și comentariile să înceapă în aceeași coloană, exceptând cazurile în care întreg rândul este un comentariu.
3. Folosiți comentarii pentru a explica ce reține fiecare registru și 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 desfășurarea 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    ;  $R3 \leftarrow R3 + R2$ 
ADD      R1, R1, #-1   ; R1 păstrează numărul de iterații (de
                        însumări care trebuie făcute)  $R1 \leftarrow R1 - 1$ 
```

[illegible]

NUMBER

- ; pentru ca programul să execute înmulțirea dintre numerele dorite trebuie ca
- ; la locația NUMBER să se intervină la nivelul simulatorului și 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. CORESPONDENȚA LIMBAJ DE ASAMBLARE LC-3 – LIMBAJ MAȘINĂ 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 mașină, biții 15÷12 reprezentând **opcode-ul** (se reamintește faptul că LC-3 ISA conține 16 instrucțiuni distincte).

DR – registru destinație

SR – registru sursă

Imm5 – valoare imediată pe 5 biți. Instrucțiunile de adunare și „ȘI Logic” pot opera atât cu doi operanzi aflați în registrii cât și cu unul situat în registru și 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 și mod de adresare indexat iar **Index6** reprezintă deplasamentul pe 6 biți.

Pageoffset9 – 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 regiștrii 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 oprește execuția și se afișează un mesaj corespunzător pe consolă.
IN	TRAP x23	Se afișează un mesaj pe consolă – „gen prompter”, citește (cu ecou) un caracter de la tastatură.. Codul ASCII al caracterului respectiv este memorat în registrul R0[7:0].
OUT	TRAP x21	Afișează pe consolă caracterul având codul ASCII în R0[7:0].
GETC	TRAP x20	Citește (fără ecou) un caracter de la tastatură.. Codul ASCII al caracterului respectiv este memorat în registrul R0[7:0].
PUTS	TRAP x22	Afișează pe consolă un șir de caractere încheiat cu terminatorul NULL. Adresa șirului se găsește în R0.

Tabelul 7.2. Servicii Sistem

7.3. PROCESUL DE ASAMBLARE – ETAPELE GENERĂRII CODULUI MAȘINĂ

Procesul de asamblare presupune convertirea unui fișier scris în mnemonică de asamblare (.asm – la LC-3 și Intel, .s la MIPS) într-un fișier executabil pe platforma destinație (în cazul de față 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 funcțiilor de bibliotecă apelate în modulul respectiv sau cu alte fișiere obiect. Se disting două etape:

- Scanarea codului sursă asamblare și **generarea tabeli de simboluri**

- Reanalizarea programului sursă și folosind informațiile din tabela de simboluri se **generează codul în limbaj mașină specific mașinii destinație**

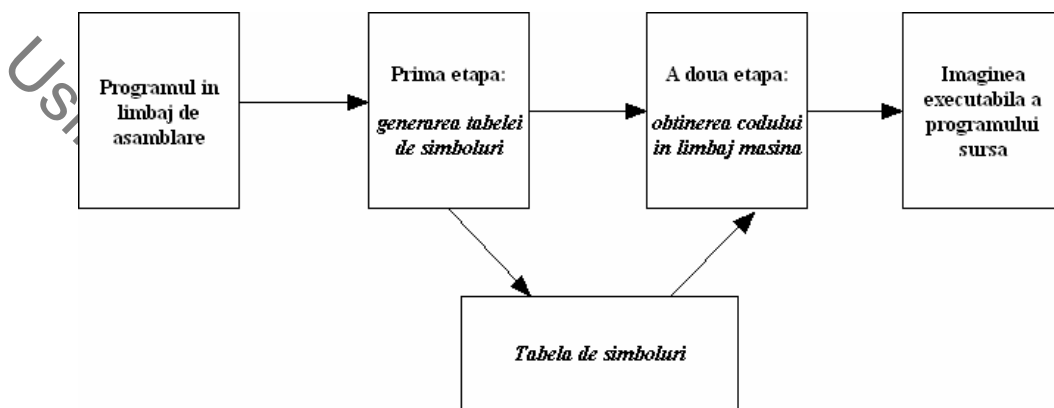


Figura 7.2. Etapele generării codului mașină pornind de la limbajul de asamblare LC-3

7.3.1. GENERAREA TABELEI DE SIMBOLURI (TS)

Reprezintă prima etapă a procesului de asamblare și presupune următoarea secvență de pași:

- Se parcurge fiecare linie din fișierul sursă asamblare.
- Pentru fiecare linie nevidă din program, se determină dacă aceasta conține o etichetă (pe prima coloană) și se calculează adresa corespunzătoare (vezi mai jos) adăugându-se la tabela de simboluri. O linie care conține doar comentariu se consideră vidă din punctul de vedere al asamblorului.
- Se identifică directiva **.ORIG**, care stabilește adresa de start a primei instrucțiuni.
- Se păstrează un contor local (LC) al adreselor (incrementat cu 1 la fiecare instrucțiune). Contorul local se inițializează 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. OBTINEREA CODULUI ÎN LIMBAJUL MAȘINĂ

Reprezintă cea de-a doua etapă a procesului de asamblare și presupune parcurgerea de la început a programului asamblare, linie cu linie, și folosind (dacă este cazul) informațiile din tabela de simboluri tocmai generată se translatează fiecare instrucțiune în cod binar (limbaj mașină). Practic fiecare opcode este înlocuit cu secvența de 4 biți corespunzătoare (vezi secțiunea 7.2.1) iar cei 8 regiștri sunt înlocuiți cu codificarea binară pe 3 biți a indexului lor. La întâlnirea unei etichete care face parte din corpul unei instrucțiuni (nu este pe prima coloană) se caută aceasta în tabela de simboluri și se ia adresa corespunzătoare. Se verifică să aparțină această adresă în intervalul $+256 / -255$ linii față de instrucțiunea în cauză (într-o pagină de 9 biți cu semn relativă la PC-ul curent). Se înlocuiește apoi în câmpul corespunzător instrucțiunii curente cei 9 biți mai puțini semnificativi ai adresei obținute făcând diferența dintre **adresa din TS și PC-ul instrucțiunii următoare** (cele 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 *immediat* care depășesc domeniul de reprezentare. De exemplu:

```
ADD    R1, R2, #1023
```

- Utilizarea adreselor (asociate etichetelor) care nu se regăsesc în aceeași pagină de memorie cu instrucțiunea curentă. În acest caz nu se poate aplica modul direct de adresare (trebuie utilizate instrucțiuni cu mod de adresare indexat).

- De asemenea pot apărea erori dacă se folosesc adrese nedeclarate extern și care nu se găsesc în tabela de simboluri.

Revenind la exemplul din subcapitolul 7.2, ținând cont și de tabela de simboluri generată în subcapitolul 7.3.1, codul în limbaj mașină 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 fișier sursă asamblare (.asm), folosind opțiunea de **Assemble** din meniul **Translate** al utilitarului **LC-3Edit**, pe lângă fișierul obiect (.obj) sunt generate următoarele fișiere auxiliare:

- Fișierul binar (.bin) conține translatarea fiecărei instrucțiuni în binar (limbaj mașină) inclusiv linia corespunzătoare directivei .ORIG.
- Fișierul binar (.hex) este copia semantică a fișierului binar cu deosebirea că liniile sunt de fapt nu valori în binar ci în hexazecimal.
- Fișierul (.sym) reprezintă tabela de simboluri (ilustrează maparea **nume_eticheta** → **Adresă**).
- Fișierul (.lst) conține pe prima coloană adresa instrucțiunii care se va procesa, pe următoarea codificarea hexazecimală a instrucțiunii, urmată de codificarea binară a instrucțiunii, numărul curent al liniei (în zecimal) și pe ultima coloană instrucțiunea în mnemonică asamblare.
- Fișierul obiect (.obj) conține aceleași informații ca și fișierul (.bin) cu deosebirea că informația (programul în limbaj mașină) în fișierul obiect este scrisă în **format binar** (date în format neprintabil) în timp ce în fișierul .bin informația este stocată în format ASCII (poate fi vizualizată folosind un editor de text gen *notepad.exe*). Fișierul .obj urmează a fi încărcat în simulatorul LC-3 și executat. Considerând un nivel suficient

de redus de abstractizare, formatul binar de fișier este caracterizat de o reprezentare compactă a informației în vederea unei analize sintactice și semantice eficiente („*parsing*”). Formatul **ELF** („*Executable and Linking Format*”) reprezintă un format binar implicit de reprezentare a fișierelor în sistemul de operare Linux. Fișierul ELF este compus dintr-un antet („ELF header”) care conține detalii despre tipul fișierului obiect – direct executabil, partajabil, relocabil, despre arhitectura țintă pe care se va procesa, etc.; antetul este urmat de 0 sau mai multe segmente și 0 sau mai multe secțiuni. Segmentele conțin informații necesare pentru execuția fișierului de aplicație, în timp ce secțiunile conțin date importante necesare procesului de creare a legăturilor între mai multe fișiere obiect (link-editare) și eventual pentru relocare (vezi în continuare).

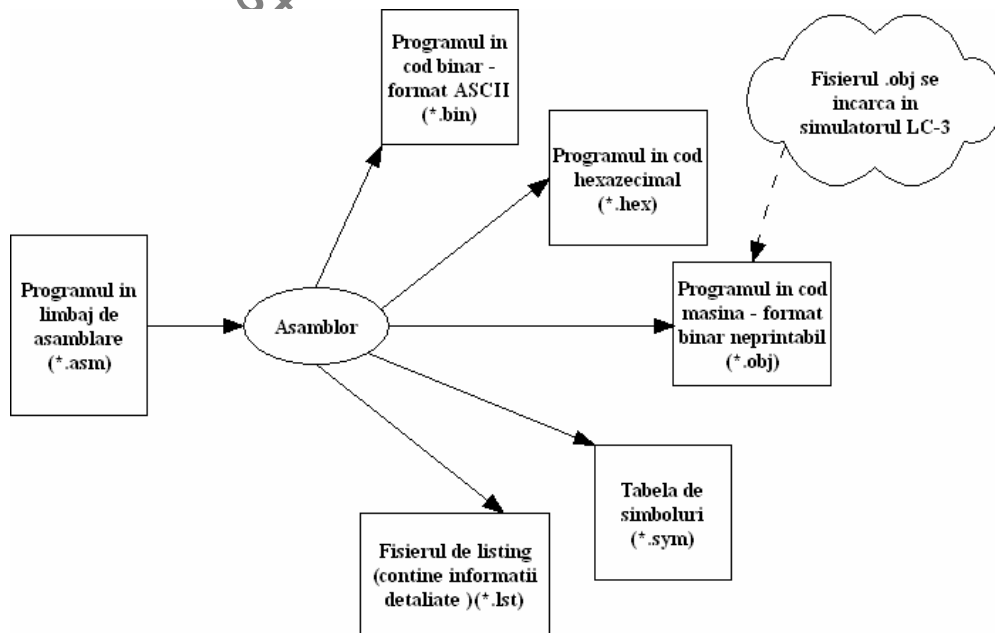


Figura 7.3. Fișierele generate de asamblorul inclus în LC-3Edit în urma asamblării unui cod sursă

7.3.3. CREAREA LEGĂTURILOR ÎNTRE MODULE, ÎNCĂRCAREA ȘI LANSAREA ÎN EXECUȚIE

Un fișier 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 și respectiv funcții de bibliotecă furnizate de către sistemul de operare. Simulatorul LC-3 permite încărcarea mai multor fișiere obiect în memorie și începerea execuției de la o anumită adresă. Un singur fișier 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 și 0xCFFF. Fiecare fișier 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` (încrucișate 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 și de încheia faza de generare de cod, pregătind încărcarea și apoi execuția.

Încărcarea și începerea execuției – „*Loading*” – reprezintă procesul de copiere a imaginii executabile a unui program în memorie (se copiază lista de instrucțiuni mașină 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 mașină să poată fi încadrat în memoria disponibilă. Problema relocării apare mai ales în situația utilizării mai multor fișiere obiect scrise de programatori independenți și 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 și adresele datelor solicitate de către instrucțiunile cu referire la memorie (load / store).

7.4. EXERCITII ȘI PROBLEME

1. Secvența de program de mai jos urmărește să deplaseze registrul R3 la stânga cu 4 biți, dar conține o eroare. Identificați eroarea și explicați cum poate fi înlăturată astfel încât programul să funcționeze 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 intenționează să citească două numere (caractere) de la tastatură, să le adune și apoi să afișeze suma.

```
ADDTHEM    TRAP    x23      ; IN – se citește primul număr
           ADD     R4,R0,#0
           TRAP    x23      ; IN – se citește al doilea număr
           ADD     R0,R4,R0
           TRAP    x21      ; OUT – afișare rezultat
           TRAP    x25      ; HALT – încheiere program utilizator
```

- a) Ce nu este în regulă cu acest program ?
b) Dacă se introduce de la tastatură 5, și apoi 3, ce rezultat se va afișa pe consolă (ecran) ?

3. Câte locații de memorie va aloca asamblorul LC-3 pentru stocarea următorului mesaj? (toate numerele din soluție sunt în sistemul de numerație 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[0xB000] = 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 șir de caractere este sau nu **palindrom**. Un palindrom este un șir de caractere care citit invers (de la dreapta la stânga) este identic cu șirul citit în sensul normal (de la stânga la dreapta) Un astfel de exemplu este șirul “racecar”. Presupunând că șirul este stocat în memorie începând cu adresa 0x4000, și se încheie cu valoarea 0x0, similar cu funcția directivei de asamblare .STRINGZ. Dacă șirul 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 și executat. Din fericire, nu există erori depistate nici în faza de asamblare nici în cea de execuție. Ce se afișează pe ecran în urma execuției? Presupunem că toți regiștrii sunt inițializați cu 0 înainte de începerea execuției. Se reamintește că întreruperea software TRAP 0x22 afișează pe ecran un șir de caractere, a cărui adresă se află în registrul R0, până întâlnește terminatorul NULL.

LABEL

- D

```
LD R1, A
AND R2, R1, #1
BRp B
```

- ADD R1, R1, #-1

- ```
ADD R0, R0, R1
ADD R1, R1, #-2
```

- BRp B
- 
- ST R0, C
- 
- TRAP x25

- .BLKW 1

- ```
.BLKW 1
.END
```

Eticheta (simbol)	Valoare

[illegible]

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 și depune rezultatul în R1. De exemplu, dacă R0 reține valoarea 0001001101110000, atunci după execuția programului rezultatul stocat în R1 va fi 00000000000000110.

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 și 30000 (zecimal). Secvența se încheie când se întâlnește 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 întâlnește 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 șiruri de caractere de aceeași lungime. Definiția celor două șiruri (stocarea lor în memorie) s-a realizat folosind directiva de asamblare .STRINGZ *șir*. Primul șir începe de la locația FIRST din memorie, iar al doilea de la locația SECOND. Dacă cele două șiruri sunt identice, programul se încheie setând valoarea 0 în R5. Dacă șirurile 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 SUBROUTINE – DIRECTE ȘI INDIRECTE. REVENIRI. SALVAREA ȘI RESTAURAREA REGISTRIILOR. STRATEGIILE “*CALLER-SAVE*” RESPECTIV “*CALLEE SAVE*”

8.1. ÎNTRERUPERI. DEFINIȚIE. CLASIFICARE

Prin **întrerupere** se înțelege oprirea programului în curs de execuție și 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 general¹¹ pe prima instrucțiune de după cea pe care a apărut întreruperea.

Din punct de vedere al generării lor, întreruperile se clasifică în **hardware** și **software**. Întreruperile inițiate hardware apar ca răspuns la un semnal extern, fiind de două tipuri: **nemascabile** (sau nedezactivabile) și **mascabile** (sau deactivabile). 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 și 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, depășire 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 aceeași 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 aceleași condiții de intrare, în timp ce întreruperile hardware sunt de obicei independente de execuția procesului curent.

8.2. ÎNTRERUPERI SOFTWARE LA NIVEL *LOW* – INSTRUȚIUNILE *TRAP*

La nivelul unui sistem de calcul există câteva instrucțiuni prin care un anumit program poate afecta comportamentul altuia (numite *privilegiate*) și care sunt cel mai bine executate de către un program *supervizor* (sistemul de operare) și nu de către programele *utilizator*.

- ♦ Instrucțiuni de Intrare / Ieșire care necesită cunoștințe specifice de utilizare și protecție a regiștrilor de interfață cu dispozitivele periferice (KBDR, KBSR – tastatura și CRTSR, CRTDR – monitorul, etc). Întrucât dispozitivele periferice sunt partajate între mai multe programe și mai mulți utilizatori, o gresală 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 și cedarea controlului sistemului de operare.
- ♦ Încărcarea regiștrilor aferenți dispozitivelor periferice și 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 și dispozitivele periferice. Aceștia folosesc aceleași instrucțiuni de transfer date utilizate la citirea / scrierea datelor din / în memorie în / din regiștrii 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 și 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

Înteruperile 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 regiștrii 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 numește *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 numește instrucțiune de intrare. Similar, o instrucțiune Store a cărei adresă destinație este specificată printr-un registru aferent unui dispozitiv periferic de ieșire din sistem, se numește instrucțiune de ieșire. Întrucât sunt folosite aceleași tipuri de instrucțiuni de acces la memorie și pentru periferice, regiștrii dispozitivelor de intrare / ieșire 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 **regiștrii de intrare / ieșire sunt mapați spre o zonă de adrese specifică dispozitivelor periferice și nu memoriei de date**. Exemple de familii de procesoare care folosesc regiștrii aferenți dispozitivelor periferice mapați în memorie în combinație cu instrucțiuni load / store sunt MIPS, SPARC și Motorola, iar Intel folosește instrucțiuni speciale de intrare / ieșire incluse în ISA (IN / OUT) pentru citirea sau scrierea datelor din / în regiștrii dispozitivelor periferice.

- Sunt parte integrantă a sistemului de operare; adresele de început a fiecărei rutine sunt arbitrar alese în spațiul de memorie $0x0000 \div 0x2FFF$ sau $0xCFFF \div 0xFFFF$.
- Pot fi implementate până la 256 de rutine la arhitectura LC-3 ISA.
- Rutinele de serviciu asigură trei funcții speciale:
 - Izolează programatorii de detalii specifice sistemului de operare.
 - Permite scrierea codului frecvent utilizat o singură dată și 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 $0x0000$ și până la adresa $0x00FF$, numită **tabela vectorilor de întrerupere TRAP** sau blocul de control sistem în cazul altor arhitecturi. Deși în versiunea actuală a arhitecturii LC-3 ISA sunt implementate doar 5 apeluri sistem, orice tentativă de folosire (scriere / citire) a zonei de adrese $0x0000 \div 0x001F$ va genera o excepție.

3. Instrucțiunile TRAP

- Sunt folosite în programele utilizator pentru a transfera controlul sistemului de operare. Codifică în corpul instrucțiunii un vector de întrerupere pe 8 biți (codul apelului sistem) care reprezintă un index în tabela cu adresele de start a rutinelor de serviciu. Informația de la locația specificată reprezintă adresa primei instrucțiuni din rutina de tratare a întreruperii, valoare care se încarcă în registrul PC.

4. Instrucțiunea 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 numește *IRET*, iar la MIPS se numește *rfe - return from exception*). Revenirea se face pe instrucțiunea imediat următoare instrucțiunii TRAP.



Figura 8.1. Formatul instrucțiunii TRAP

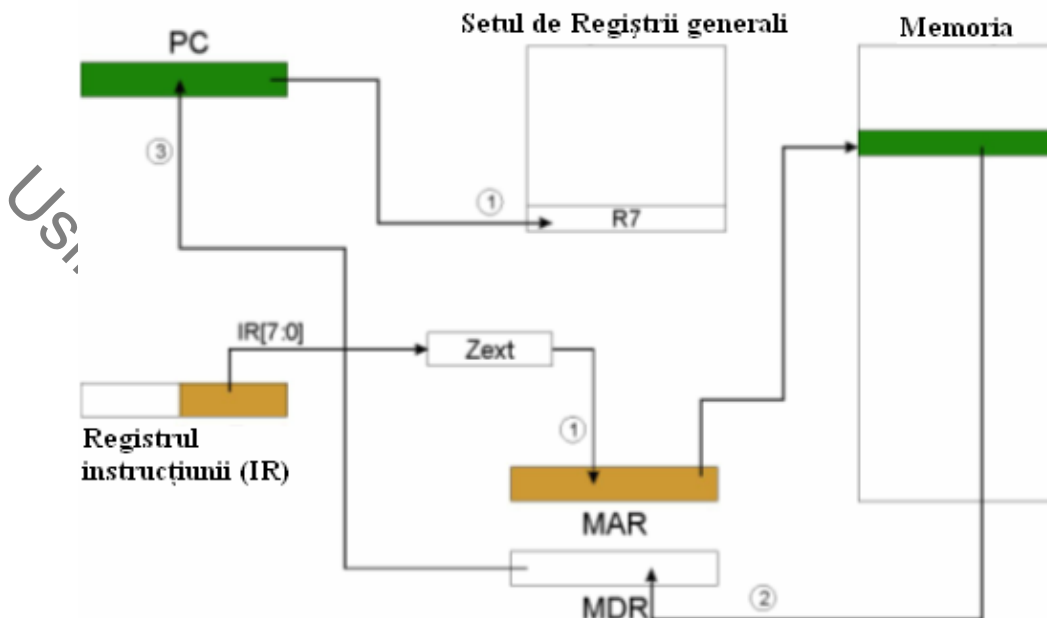


Figura 8.2. Modul de lucru al instrucțiunii TRAP

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 și 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 regiștrii (la MIPS – \$ra și \$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

conținutul registrului R7 în registrul PC cedând astfel controlul înapoi programului apelant (utilizator) pe instrucțiunea imediat următoare instrucțiunii TRAP (doar dacă nu cumva a fost alterată în interiorul rutinei). Formatul instrucțiunii RET este cel din figura 8.3 iar modul său de lucru este ilustrat în figura 8.4.



Figura 8.3. Formatul instrucțiunii RET

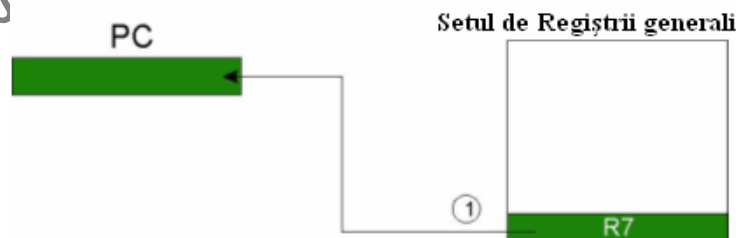


Figura 8.4. Modul de lucru al instrucțiunii RET

Vector	Simbol	Operații efectuate
x20	GETC	<ul style="list-style-type: none"> Citește un singur caracter de la tastatură fără ecou. Scrie codul ASCII al caracterului citit în registrul R0[7:0], biții R0[15:8] setându-i pe 0.
x21	OUT	Afișează pe monitor caracterul al cărui cod ASCII se regăsește în registrul R0[7:0].
x22	PUTS	Afișează pe consolă șirul de caractere al cărui adresă de început se află în registrul R0.
x23	IN	<ul style="list-style-type: none"> Afișează un mesaj pe consolă și citește fără ecou un caracter de la tastatură. Înscrie codul ASCII al caracterului citit în R0[7:0] și resetează biții R0[15:8].
x25	HALT	Afișează un mesaj de atenționare pe ecran și întrerupe execuția programului.

⋮	⋮
x0020	x0400
x0021	x0430
x0022	x0450
x0023	x04A0
x0024	x04E0
x0025	xFD70
⋮	⋮

Figura 8.5. Semantica instrucțiunilor TRAP. Adresele de start aferente rutinelor de tratare ale întreruperii

Se consideră următorul exemplu care folosește apelurile sistem de citire / scriere caracter și încheiere program. Problema care se rezolvă citește un caracter de la tastatură atâta timp cât este diferit de cifra 7 și 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 și mari
AGAIN:	TRAP x23;	Se citește 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 ieșire din program
	ADD R0, R0, R3;	Se adaugă diferența de 0x20 la codul ASCII al caracterului introdus de la tastatură
	TRAP x21 ;	Se afișează 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 și 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 afișare caracter – **Output Service Routine (TRAP 0x21)**.

	.ORIG x0430 ;	Adresa de start a rutinei de tratare a apelului TRAP 0x21
	ST R7, SaveR7;	Salvează conținutul regiștrilor R7 și R1 – posibil a fi modificați pe parcursul rutinei de tratare
	ST R1, SaveR1	
	; ----- Scrie caracter	
TryWrite	LDI R1, CRTSR;	Citește starea monitorului (registrul CRTSR). Dacă bitul cel mai semnificativ este 1 atunci se înscrie codul ASCII al caracterului de afișat 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 afișat în registrul de date al monitorului (registrul CRTDR)
	; ----- Revenire din apelul sistem de afișare TRAP 0x21	
Return	LD R1, SaveR1;	Restaurează conținutul regiștrilor R7 și R1 care au fost modificați 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	

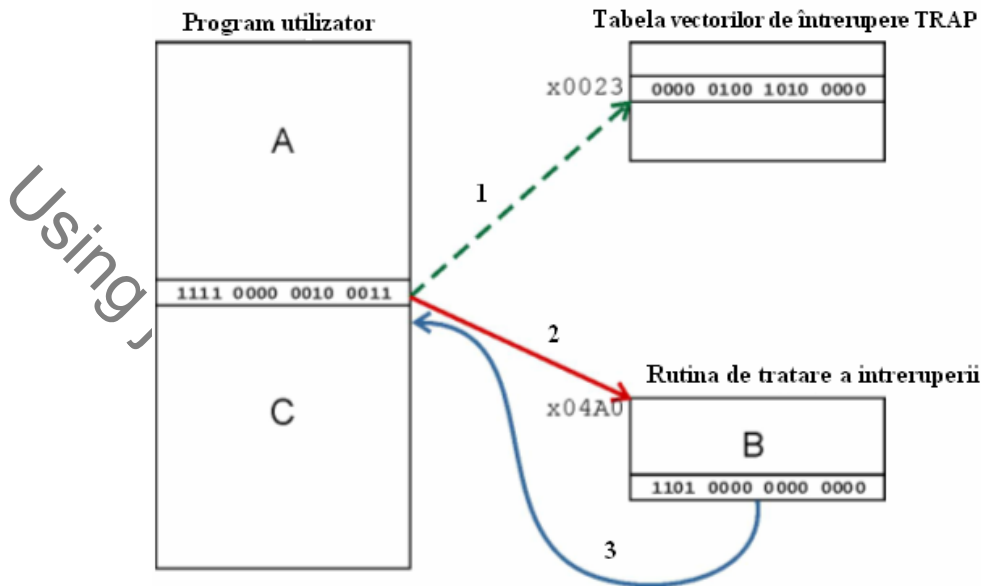


Figura 8.6. Etapele parcurse în urma apariției unui apel sistem de citire caracter în programul utilizator

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 și până la revenirea din rutina de tratare. Pe scurt, în rutina de tratare se desfășoară următoarele subetape: se afișează caracterul special *newline* (deci implicit sunt incluse instrucțiunile de afișare caracter), urmat de afișarea unui mesaj care solicită introducerea unui caracter de la tastatură (sunt incluse instrucțiunile de afișare șir de caractere). Urmează citirea propriu-zisă de la tastatură iar apoi afișarea ecoului și în final trecerea pe o nouă linie.

```
.ORIG x04A0
; ----- Salvează conținutul regiștrilor care vor fi alterați la nivelul subrutinei
START      ST    R7, SaveR7    ; Salvează adresa de revenire în
                                programul apelant (R7)
            ST    R1, SaveR1    ; Salvează valorile regiștrilor necesari
                                în rutina: R1, R2 și R3
            ST    R2, SaveR2
            ST    R3, SaveR3    ; trebuie restaurați înaintea execuției
                                instrucțiunii RET
```

; ----- Scribe caracter special **newline**

	LD	R2, Newline	; Se încarcă în R2 codul ASCII al caracterului special <i>newline</i>
L1	LDI	R3, CRTSR	; Verifică starea perifericului – dacă permite scrierea în registrul de date al monitorului <i>CRTDR este disponibil?</i>
	BRzp	L1	; Registrul CRTDR este disponibil dacă bitul cel mai semnificativ al registrului de stare (cel de semn – CRTSR ₁₅) este 1. Dacă nu este încă disponibil pentru afișare atunci se va continua cu citirea stării acestuia.
	STI	R2, CRTDR	; Este permisă afișarea caracterului <i>Newline</i> la consolă

; ----- Afișează pe ecran mesajul de atenționare pentru citirea unui caracter de la tastatură

	LEA	R1, Prompt	; Prompt reprezintă adresa de start a mesajului de atenționare
Loop	LDR	R0, R1, #0	; Încarcă în R0 caracter după caracter din mesajul de afișat
	BRz	Input	; Dacă s-a ajuns la sfârșitul mesajului trece la citirea caracterului de la tastatură
L2	LDI	R3, CRTSR	; Verifică starea monitorului
	BRzp	L2	
	STI	R0, CRTDR	; Periferic disponibil ⇒ afișează caracterul curent al mesajului
	ADD	R1, R1, #1	; Trece la următorul caracter din mesaj
	BRnzp	Loop	; Repetă afișarea până la sfârșitul șirului

; ----- 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 – KBSR ₁₅) este 1. Dacă nu este încă disponibil pentru

afişare atunci se va continua cu citirea stării acestuia.

```

LDI    R0, KBDR    ; Preia caracterul apăsător 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 conţinutul regiștrilor care au fost alteraţi la nivelul subrutinei
        LD     R1, SaveR1    ; Preluarea lui R1, R2 și R3 de la
                            ; adresele unde au fost salvaţi
        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 instrucţiune în programul
                            ; apelant ulterioară TRAP 0x23
                            ; (echivalentă cu: JMP R7)
; ----- Spaţiul 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 <i>newline</i>
Prompt	.STRINGZ "Input a character>"	; mesaj de atenționare
	.END	; încheierea secvenței de cod

Se reamintește că oprirea sistemului de calcul presupune oprirea unității de control, deci anularea semnalului de tact care reprezintă „*pulsul*” sistemului (vezi capitolul 5, figura 5.8). Arhitectura setului de instrucțiuni aferentă primelor sisteme de calcul conțineau o instrucțiune specială (HALT) pentru oprirea sistemului. Ulterior, proiectanții acestora au realizat cât de rar este folosită instrucțiunea HALT și totodată, faptul că se irosește un cod de operație pentru stocarea ei, care uneori poate fi necesar pentru definirea altor instrucțiuni (operații) cu caracter mult mai frecvent. La ora actuală, sistemele de calcul folosesc apeluri (rutine) sistem pentru anularea semnalului de tact și implicit oprirea unității de control. În cadrul LC-3 ISA, prin întreruperea TRAP 0x25 se urmărește practic oprirea ceasului sistemului, însă, aflându-ne într-un simulator (aplicație software) și nu pe un calculator gazdă, practic se oprește funcționarea simulatorului LC-3 (a aplicației care a rulat pe acesta) și se predă controlul sistemului de operare care rulează și gestionează procesele aferente calculatorului gazdă.

Secvența următoare de cod reprezintă conținutul rutinei de tratare aferentă apelului sistem de încheiere program și transfer control sistemului de operare – **Halt Service Routine (TRAP 0x25)**. Pe scurt, în rutina de tratare se desfășoară următoarele subetape: se afișează caracterul special *newline* (folosindu-se apelul sistem TRAP 0x21), urmat de afișarea unui mesaj care atenționează asupra încheierii aplicației utilizator și cedarea controlului sistemului de operare (se folosește apelul sistem TRAP 0x22 pentru afișarea șirului de caractere). Urmează apoi resetarea bitului cel mai semnificativ al registrului de control al mașinii (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 mașinii (MCR)
ST	R0, SaveR0	; R0 este folosit ca temporar în operațiile din rutina de serviciu

; ----- *Afișează pe ecran mesajul de atenționare 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 mașinii (MCR_{15}) pentru oprirea ceasului sistemului*

```
LDI   R1, MCR    ; Încarcă conținutul registrului de
                  ; control al mașinii în R1. Registrul –
                  ; fiind reprezentat de o locație de
                  ; memorie – este necesar accesul
                  ; indirect.
LD     R0, MASK   ; R0 = 0x7FFF, reprezintă masca
                  ; pentru ștergerea 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 regiștrilor alterați în
                  ; rutină. Preluarea lui R0 și 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 secvenței de cod

8.2.2. SALVAREA ȘI RESTAUREA REGIȘTRILOR ÎN CAZUL RUTINELOR DE SERVICIU

Conținutul unui registru trebuie salvat dacă registrul va fi modificat de către rutina de serviciu iar valoarea acestuia va fi necesară și după încheierea rutinei. Se pune întrebarea cine salvează valoarea registrului:

- **Apelantul** rutinei de tratare (programul utilizator) care știe ce regiștrii sunt necesari după încheierea rutinei, dar nu știe ce regiștrii sunt modificați în rutină? Salvările efectuate la nivelul apelantului poartă numele de „**caller-save**” și presupune stocarea temporară în memorie a conținutului regiștrilor care sunt distruși de către programul apelant sau în rutinele apelate (dacă este vizibil sau apriori cunoscut ceea ce se modifică în rutină) și valorile acestora sunt necesare ulterior. În cazul arhitecturii LC-3 ISA, strategia „**caller-save**” impune salvarea lui R7 înaintea oricărei instrucțiuni TRAP și 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 regiștri generali ai procesorului, presupune evitarea folosirii acestor regiștrii cu funcții speciale (adresă de revenire, indicator de stivă, registrul care păstrează valoarea returnată din (sub)rutine) în operații uzuale (aritmetico-logice, etc).

sau

- **Rutina apelată** (în acest caz cea de tratare a întreruperii)? Avantajul acestei variante este că, rutina cunoaște regiștrii pe care îi va modifica, dar nu cunoaște ce regiștrii vor fi necesari ulterior după execuția rutinei. În acest caz, strategia de salvare a regiștrilor se numește „**callee-save**”. Înainte de intrarea propriu-zisă în (sub)rutină sunt salvați toți regiștrii care vor fi alterați, exceptându-i pe cei care, chiar dacă sunt alterați, nu sunt necesari la nivelul programului apelant (totuși, acest lucru nu este întotdeauna vizibil de la nivelul (sub)rutinei apelate). Înainte de revenirea din (sub)rutină, regiștrii salvați inițial sunt restaurați la valorile anterioare apelului. Opțiunea „**callee-save**”

este utilizată în mod frecvent și în cazul apelurilor de subrutine (rutine definite de programator / utilizator) nu doar în situația rutinelor de tratare a întreruperilor.

Salvarea și restaurarea regiștrilor 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 funcției etc.) este „vizibil” sau accesibil într-un anumit loc. Astfel, locul în care o entitate este vizibilă este referit ca **scopul** entității.

În continuare se prezintă un exemplu care dovedește necesitatea salvării registrului R7 (adresa de revenire) de către programul apelant înaintea execuției unui apel sistem (*TRAP 0x23* în acest caz). Programul urmărește 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. Greșeala este provocată de faptul (ascuns la prima vedere) că apelul sistem *TRAP 0x23* modifică valoarea lui R7 cu adresa instrucțiunii *ADD R0, R0, R6 (0x3004)*. La revenirea din apel R7 va avea această valoare, pozitivă, se va executa saltul la *AGAIN* și se reexecută apelul sistem, șamd, intrându-se astfel în buclă infinită.

Salvarea adresei de revenire în programul apelant trebuie făcută în cazul oricărui apel imbricat de rutină (incluzându-l și pe cel recursiv), altfel se pierde legătura spre programul apelant și revenirea se face la infinit pe aceeași instrucțiune 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 conține valoarea -48 care trebuie scăzută din fiecare caracter (cifră) citit de la tastatură pentru determinarea valorii numerice a cifrei
	LD R7, COUNT;	Inițializarea lui R7 cu 10
AGAIN	TRAP x23;	Citește 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 poziția 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 secvențe pentru ca la locația Binary să fie stocate numerele (și nu codurile lor ASCII) citite de la tastatură ?

8.3. SUBROUTINE

Subrutinele sunt secțiuni de cod (fragmente de program) definite de programator / utilizator și care:

- Sunt stocate în spațiul de memorie utilizator
- Realizează o sarcină bine precizată (execută operații bine definite de către programator)
- Este invocată (apelată) de către un alt program utilizator (sau subrutină) – program apelant
- La încheierea ultimei instrucțiuni din subrutină se predă controlul programului apelant pe instrucțiunea 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 și nu fac uz de resurse hardware protejate.

Există câteva motive în favoarea utilizării subrutinelor:

- Reutilizarea, eficiență din punct de vedere al timpului de execuție, a codului, fără a fi necesară rescrierea lui. O consecință 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 și distribuirea sarcinilor între mai mulți programatori – fiecare realizează un anumit task, independent de ceilalți. În acest fel crește lizibilitatea codului, el putând fi mai ușor de urmărit, depanat și corectat.

- Cele mai frecvent folosite funcții (subrutine) matematice, grafice, sau cu diferite caracteristici, sunt implementate și furnizate de producătorii de sisteme software, sub forma funcțiilor de bibliotecă. Se impune însă respectarea câtorva cerințe:
- Transmiterea parametrilor subrutinei apelate și returnarea rezultatului de către aceasta subrutinei apelante (eventual programului principal) se face prin regiștrii sau memorie. De exemplu, în cazul procesorului MIPS, în regiștrii \$a0, \$a1, \$a2, \$a3 sunt transmiși parametrii subrutinei apelate, iar în caz că sunt mai mulți de 4 aceștia se depun pe stivă. Rezultatul returnat de subrutine este transmis prin intermediul regiștrilor \$v0 și \$v1 dacă este necesar.
- Subrutinele trebuie să poată fi apelate din orice punct al programului (sau subrutine componente), revenirea făcându-se pe instrucțiunea imediat următoare apelului.

8.3.1. MECANISMUL DE APEL SUBROUTINE ȘI REVENIRE

Figura 8.7. ilustrează execuția unui program sursă conținând mai multe secțiuni de cod independente – secvențele contigue din punct de vedere al execuției X, A, Y, Z și W. Trebuie menționat că secvența de cod A se repetă de câteva ori astfel încât ar fi recomandabilă stocarea codului într-o subrutină, scrisă o singură dată și apelată repetat.

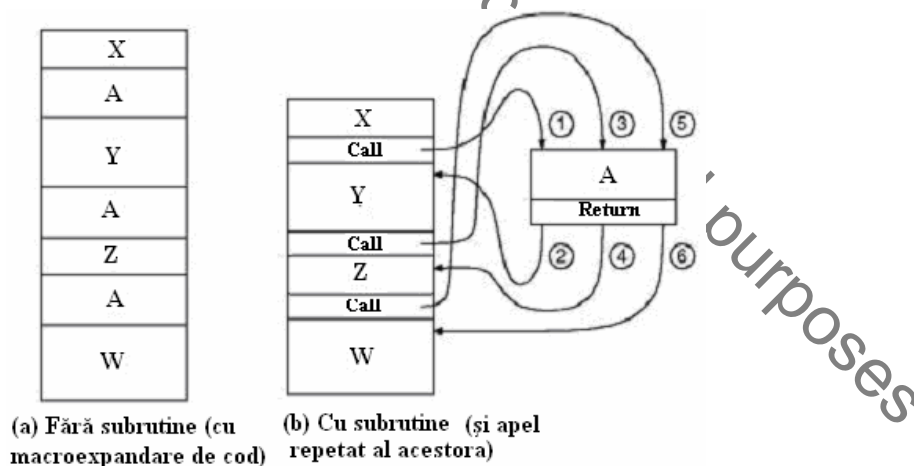


Figura 8.7. Execuția unui program sursă: (a) fără subrutine prin macroexpandarea¹³ codului, (b) folosind subrutine și apelul repetat al acestora

¹³ Repetarea (scrierea conținutului rutinei sau funcției 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 și 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 și 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 și nu un salt necondiționat) și 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 ($PC \leftarrow (PC) + \text{Sext}(\text{IR}[8:0])$).

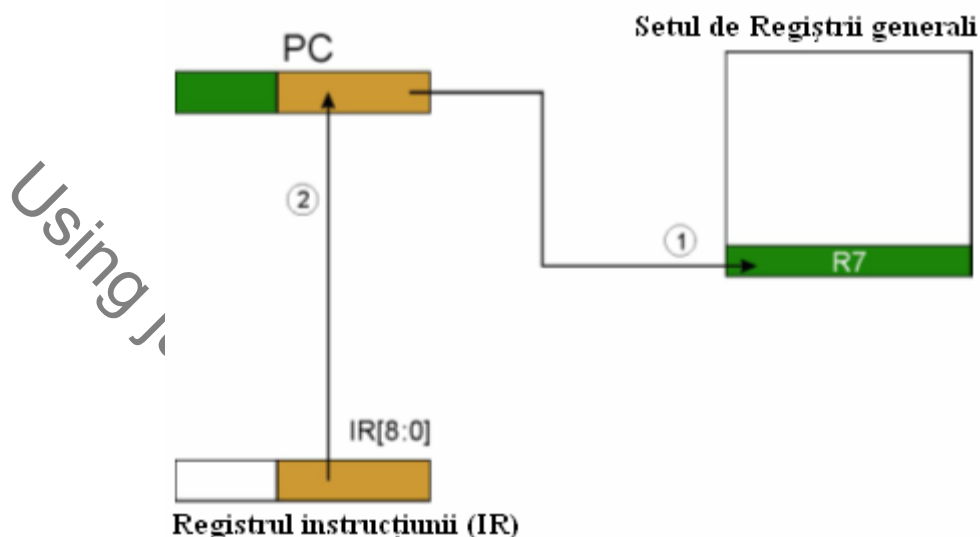


Figura 8.9. Modul de lucru al instrucțiunii JSR

În figura 8.10 este prezentat formatul instrucțiunilor JSRR / JMPR iar figura 8.11 ilustrează modul de lucru al acestora. Semantica instrucțiunilor JSRR / JMPR este similară cu cea a instrucțiunilor JSR / JMP, singura deosebire constând în faptul că adresa de apel / salt se obține 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 necondiționat implicat de instrucțiunile JSRR / JMPR să nu se realizeze neapărat într-o vecinătate a instrucțiunii curente (cazul JSR / JMP – echivalent apelurilor / salturilor de tip NEAR la procesorul INTEL) ci și în alte zone din spațiul de memorie utilizator (apeluri / salturi de tip FAR).

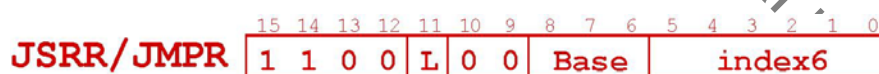


Figura 8.10. Formatul instrucțiunilor JSRR/JMPR

Primii 4 biți din formatul instrucțiunii JSRR / JMPR sunt 1100, urmați de bitul IR[11]=L cu același rol de selecție între apel (L=1) și salt necondiționat (L=0). Următorii doi biți 0 iar biții IR[8:6] codifică registrul de bază folosit în adresare. Deplasamentul este obținut din câmpul *index6* IR[5:0] extins cu 0 până la 16 biți. Adresa de apel / salt se obține prin însumarea conținutului 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 execuției instrucțiunii JSRR indică spre instrucțiunea următoare din programul apelant, PC-ul fiind incrementat în timpul fazei FETCH) – vezi operația 1 din figura 8.11. În PC se încarcă adresa primei instrucțiuni din subrutina apelată, predându-se controlul acesteia – vezi operația 3 din figura 8.11. Revenirea din subrutină în cazul instrucțiunilor JSR / JSRR se realizează cu ajutorul instrucțiunii RET, care restaurează în PC valoarea aflată în R7 (adresa primei instrucțiuni din subrutina apelantă, succesorul apelului), mecanism similar cu cel întâlnit în cazul instrucțiunilor TRAP. Întrucât în cazul instrucțiunilor JMP / JMPR nu se face apel ci doar salt necondiționat rezultă că nu se va face uz de instrucțiunea RET, în caz contrar s-ar modifica în mod eronat semantica programului (*pasul 1 din figura 8.11 nu apare în cazul instrucțiunii JMPR*).

Instrucțiunile JSRR / JMPR poartă numele de instrucțiuni de apel / salt cu mod de adresare indirect prin registru. Ele, ca de altfel toate instrucțiunile de salt condiționat / necondiționat, apel direct / indirect cauzează hazarduri de ramificație în procesarea pipeline a instrucțiunilor. Metodele de soluționare implementate în procesoarele actuale sunt fie *software* bazate pe reorganizarea codului sursă, dar mai ales *hardware* bazate pe predicția dinamică a adreselor destinație aferente instrucțiunilor de salt. După cum se știe, predicția adreselor destinație ale salturilor / apelurilor indirecte este o problemă extrem de *dificilă* (întrucât adresele țintă se modifică în mod dinamic) și *actuală* iar soluțiile propuse nu sunt încă mulțumitoare. În cadrul unei lucrări anterioare, autorul acestei cărți a sesizat faptul că, înainte de a aborda efectiv problema predicției ar trebui să înțeleagă în profunzime modurile în care aceste apeluri indirecte sunt generate prin compilare. Pentru aceasta, el dezvoltă programe de test proprii, atât procedurale cât și obiectuale, care conțin corpuri “suspectate” de a genera apeluri și salturi indirecte în urma compilării. Pe baza acestei cercetări se extrag concluzii valoroase și deosebit de utile cu privire la construcțiile din programele (limbajele) de nivel înalt (*High Level Languages*) generatoare ale unor astfel de salturi (apeluri indirecte de funcții prin pointeri, construcții *switch/case*, prezența funcțiilor de bibliotecă, inclusiv biblioteci dinamice de tip DLL, legarea dinamică realizată prin polimorfism etc.). Toate aceste informații de nivel semantic superior pot fi utilizate cu succes în predicția adreselor salturilor / apelurilor indirecte care, actualmente, utilizează în majoritatea lor informații de nivel semantic mai scăzut, precum codul obiect, adrese binare etc. Practic, pierderea semanticii codului de nivel înalt după compilare, devine inacceptabilă pentru noua generație de microprocesoare.

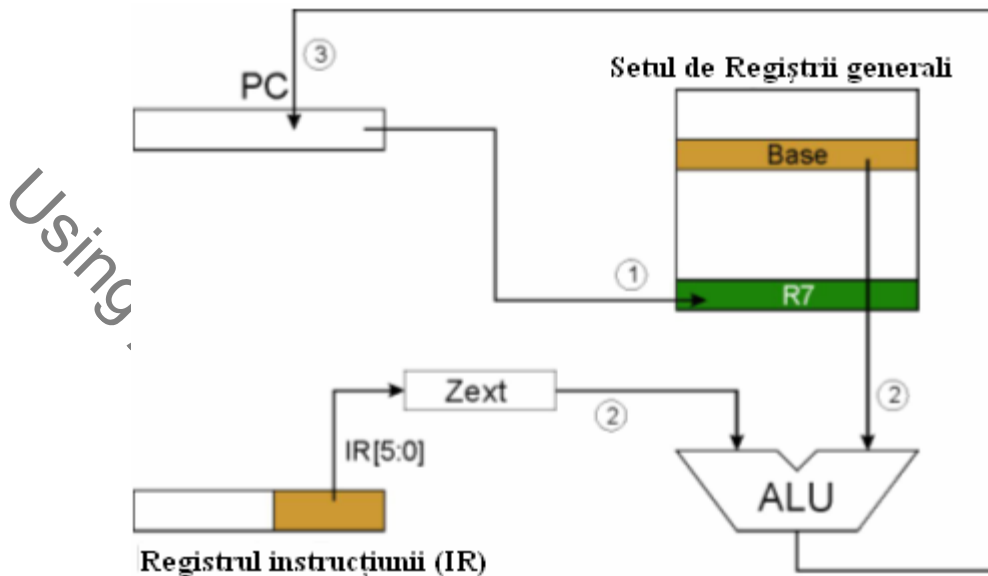


Figura 8.11. Modul de lucru al instrucțiunii JSRR

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 urmărește calcularea diferenței dintre regiștrii R1 și 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 <i>complementul față de 1</i>
		a valorii stocate inițial în R0)
	ADD R0, R0, #1	; se adună 1 pentru a determina
		<i>complementul față de 2</i>
	RET	; revenirea din subrutină

; Apelul subrutinei din programul principal trebuie făcut din aceeași pagină cu subrutina (mod de adresare direct)

; Se dorește 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 ȘI DE LA SUBROUTINE

Transferul de informație de la programul (subrutina) apelant(ă) la subrutina apelată și invers se realizează prin intermediul **argumentelor** și respectiv al **valorii returnate**.

Argumentele reprezintă una sau mai multe valori, transmise subrutinei apelate prin intermediul regiștrilor, necesare acestuia pentru a-și 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 față de 2).
- În rutina de serviciu aferentă apelului sistem OUT, R0 reprezintă codul ASCII al caracterului ce va fi afișat.
- În rutina de serviciu aferentă apelului sistem PUTS, R0 reprezintă adresa șirului de caractere ce va fi afișat.

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 față 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 instrucțiune din subrutină).
- Ce realizează subrutina (scopul, funcția, rezultatul acesteia).

Așa cum s-a amintit anterior referitor la avantajul utilizării subrutinelor și 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 mașinii (procesorului), după execuția subrutinei. De asemenea, trebuie știut argumentele și respectiv, dacă există, valoarea returnată.

8.3.3. SALVAREA ȘI RESTAURAREA REGIȘTRILOR ÎN CAZUL SUBROUTINELOR. RUTINE (FUNCȚII) DE BIBLIOTECĂ

La fel ca și în cazul utilizării rutinelor de serviciu aferente întreruperilor software, regiștrii trebuie salvați și restaurați la valorile anterioare apelului. *În aproape toate cazurile de utilizare a subrutilnelor este folosită strategia „callee-save”, excepție făcând salvarea regiștrilor care reprezintă valoarea returnată.* Sunt salvați toți regiștrii 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 și să rămână așa 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) și 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 fișiere 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 și 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. EXERCITII ȘI 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 afișează programul ? De ce sunt necesare cele două instrucțiuni care îl salvează și 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 locațiile de memorie începând cu adresa de la eticheta DATA conțin 10 numere întregi în complement față de 2. Aceste numere trebuie setate de utilizatorul programului înainte de execuție. Ce realizează programul ? Care este relația dintre valorile inițiale și cele finale de la adresele respective ?

4. Intenția următorului program este de a afișa valoarea 5 pe consolă (ecran). Din nefericire nu se întâmplă acest lucru. De ce ? Răspundeți 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. Răspundeți la întrebările:

- Care sunt diferențele dintre instrucțiunile TRAP și JSRR ? Dar dintre JSR și JSRR ?
- Există vreo instrucțiune care face același lucru ca și instrucțiunea JMP ?
- Care este scopul apelurilor sistem ?
- Care este diferența dintre strategiile “*caller-save*” și “*callee-save*” în salvarea și restaurarea regiștrilor la apelul și revenirea din subrutine.
- Ce anume (dacă există ceva) este greșit în următoarea subrutină ?

```

DO      JSR DOTHIS
      JSR DOTHAT
      RET
    
```

6. În tabelul de mai jos sunt ilustrate conținuturile regiștrilor arhitecturii LC-3 înainte și după execuția instrucțiunii de la adresa 0x3010. Identificați instrucțiunea memorată la adresa 0x3010. De notat că, există suficientă informație în tabelul specificat care să personalizeze instrucțiunea.

	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:

x3010: 15 0

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

7. Cele două secvențe de mai jos sunt asamblate separat. În urma acestui proces sunt detectate două erori: una în faza de asamblare și una în faza de link-editare. Încercuiți fiecare greșeală și stabiliți în ce fază a fost detectată.

<pre> SQRT .EXTERNAL .ORIG 0x3000 LDR R0, VALUE JSR SQRT ST R0, DEST HALT VALUE .FILL 0x30000 DEST .FILL 0x0025 .END </pre>	<pre> SQRT .ORIG 0x3200 ADD R0, R0, #0 ;codul necesar implementării funcției de extragere a rădăcinii pătrate ;returnează rezultatul în R0 RET .END </pre>
---	---

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	

; negația (valoarea
în complement
față de 2) a
codului ASCII
pentru tasta
"Enter"

Tabela de Simboluri

[illegible]

b) Ce realizează programul anterior ? Explicați rolul fiecărui registru folosit: R0, R1, R3 și R6. Mai există și alți regiștrii care își 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 își introducă numele. Programul memorează mesajul „Hello, ” urmat de numele introdus, ca un șir de caractere începând de la adresa indicată de simbolul HELLO și-l afișează 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
<hr/>			
AGAIN2	TRAP	0x20	; GETC
	TRAP	0x21	; OUT
	ADD	R2, R0, R3	
	BRz	CONT	
<hr/>			
	BRnzp	AGAIN2	
CONT	AND	R2, R2, #0	
<hr/>			
	LEA	R0, HELLO	
	TRAP	0x22	; PUTS


```

                                TRAP      0x25                ; HALT
NEGENTER      .FILL      0xFFFF6                ; -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 afișa pe ecran mesajul: ABCFGH. Completați instrucțiunile lipsă astfel încât programul original să realizeze ceea ce și-a propus. În fiecare spațiu lipsește 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 știința și 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*. Definiție pentru stivă reprezintă modul de acces la elementele sale și nu modul specific de implementare. Din punct de vedere software stiva este o structură de date abstractă definită prin regulile de inserare și extragere a datelor în / din ea, ambele operații fiind efectuate la același 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, și că primul introdus în stivă este ultimul care va fi scos.

9.1.2. IMPLEMENTARE HARDWARE ȘI 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. Și într-un caz și 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.

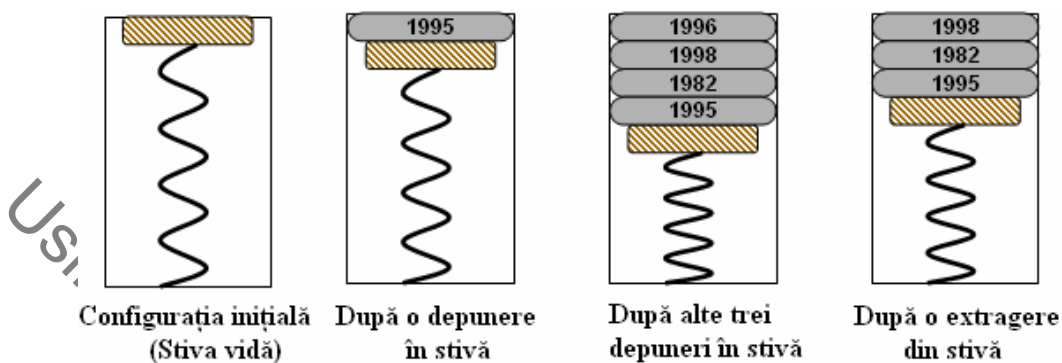


Figura 9.1. Exemplu de stivă. Model fizic

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 (necomprimit). Prima depunere în stivă este cea a monedei de etichetă 1995. Urmează alte trei depuneri: întâi moneda etichetată 1982, apoi cea 1998 și 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 regiștrii. În acest caz, stiva este alcătuită dintr-un număr de regiștrii 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 regiștrii implică deplasarea informației între regiștrii inacceptabil din punct de vedere al memoriei implementată fizic** (este inefficient 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 regiștrii). 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 și 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 și apoi 5). Trebuie observat că vârful stivei (TOP – *top of the stack*) – indicator la ultima locație ocupată din stivă, rămâne tot timpul în același loc. Întrucât în stivă mai rămân valorile 31 și 5 funcția Empty va genera Nu.

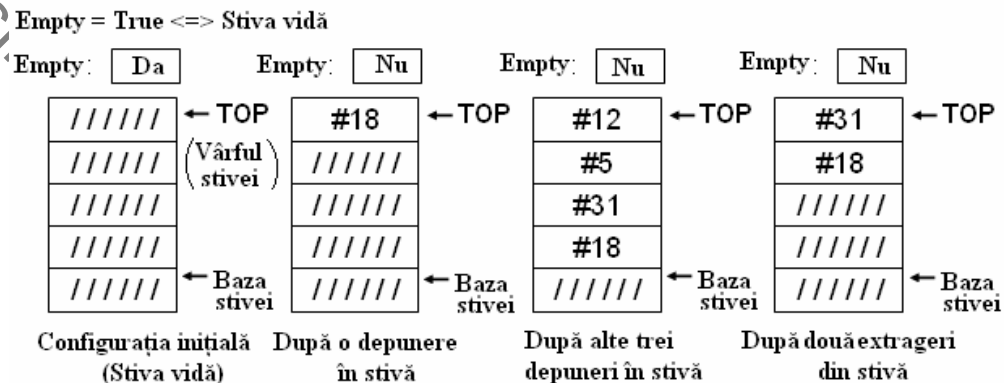


Figura 9.2. Implementarea stivei de monede în hardware folosind regiștrii

Implementarea software a stivei are la bază ideea de a nu permite deplasarea informației în memorie la fiecare acces la stivă. Doar vârful stivei poate să-și modifice poziția, el fiind reținut într-un singur registru.

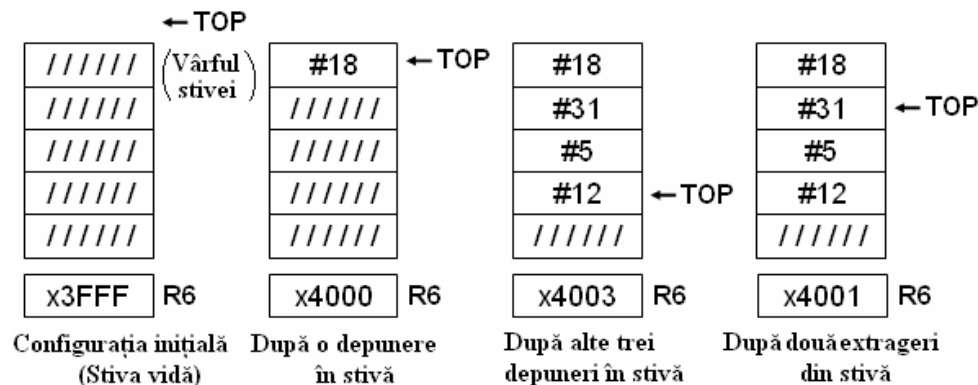


Figura 9.3. Implementarea software a stivei

Prin convenție la LC-3 ISA [Patt03] registrul R6 reține adresa vârfului stivei (reprezintă un indicator spre ultima locație ocupată din stivă). La alte arhitecturi (Intel, MIPS) acest registru se numește SP (*stack pointer* – vezi capitolul 12 pentru clarificarea noțiunii 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 și 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 și 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 și 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) și de extragere din stivă (Pop). De menționat că, procesoarele Intel au definite instrucțiuni cu același nume *PUSH reg* și *POP reg*. Procesoarele RISC însă (denumite și mașini load/store) implementează operațiile cu stiva prin instrucțiuni de scriere și citire din memorie (*store* – pentru Push) și (*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 ; citește din memorie data de la adresa specificată de TOS
ADD R6, R6, #-1 ; decrementează indicatorul de stivă (se modifică TOS).

Obs: De remarcat că, deși în cazul unei stive fizice (de farfurii – de exemplu), la extragerea din stivă elementul dispăre, 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 operația ce marchează extragerea unui element din stivă permițându-se în continuare încărcarea altuia la locația „recent eliberată”.

Condițiile 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 apariției acestor condiții de eroare trebuie monitorizată adresa vârfului stivei (top of stack – *TOS*). În cadrul LC-3 ISA, vârful stivei reține tot timpul adresa ultimei locații ocupate din stivă (fiecare operație Push incrementează *TOS* iar fiecare operație Pop îl decrementează). Astfel înaintea fiecărei operații de depunere în stivă trebuie întâi actualizat vârful stivei și apoi scrisă informația la noua adresă (noul *TOS*) iar la fiecare extragere din stivă trebuie, după citirea datei din stivă, să se actualizeze *TOS*. Stiva crește în capacitate (în locații utilizate) în jos. La anumite procesoare (MIPS pe platformă Linux) stiva crește în capacitate de la adrese mari spre adrese mici (deci registrul SP va fi decrementat la fiecare operație Push și incrementat la fiecare Pop).

Practic înaintea operației de depunere în stivă trebuie testată condiția de *overflow* iar la extragere trebuie testată condiția de *underflow*. Schema logică de mai jos verifică dacă înaintea unei extrageri stiva este vidă și marchează în registrul R5 rezultatul (1 dacă apare *underflow* și 0 dacă se poate face extragerea, iar în R0 se depune valoarea din vârful stivei).

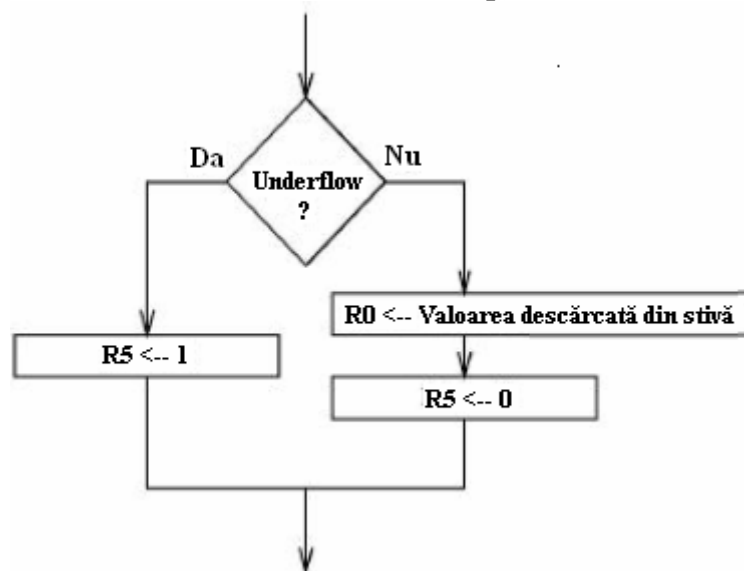


Figura 9.4 Testarea condiției de *underflow*

Astfel prin introducerea instrucțiunilor care testează condițiile de eroare codul echivalent instrucțiunilor *PUSH* și *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) și marchează în registrul *R5* rezultatul (1 dacă apare *overflow* și 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 facilitățile mecanismului de stivă o reprezintă posibilitatea apelării funcțiilor imbricate (recursivitate directă sau indirectă) și după cum se va vedea în continuare al tratării întreruperilor imbricate (aparitia unei întreruperi în chiar rutina de serviciu (tratare) a altei întreruperi, atunci când este permis). După cum s-a mai specificat și în capitolul 8, o întrerupere hardware poate fi considerată un apel de subrutină (spre necunoscut din punct de vedere al programatorului dar nu și al procesorului și al sistemului de operare) declanșat asincron de un eveniment extern.

Modul de lucru prin întreruperi hardware urmărește tratarea eficientă din punct de vedere al performanței de procesare a interacțiunilor dintre dispozitivele periferice și procesor. De câte ori dorește un serviciu de la procesor, dispozitivul extern, prin interfața sa, lansează un semnal de întrerupere (de exemplu, când se introduce un caracter de la tastatură sau când monitorul / imprimanta sunt gata pentru a afișa un nou caracter). În urma recepționării semnalului de întrerupere, procesorul încheie ciclul instrucțiunii aflate în execuție și părăsește programul principal urmând să intre în rutina de serviciu a întreruperii respective (similară rutinelor de serviciu aferente întreruperilor software – instrucțiunilor 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 instrucțiunii RTI la LC-3 ISA, IRET la Intel sau JR \$31 la MIPS), reluarea procesării făcându-se cu instrucțiunea 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 și 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 și un dispozitiv extern. Când respectivul periferic vrea să întrerupă activitatea procesorului și 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 așa numitul **protocol hardware de acceptare a întreruperii** desfășurat 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 și faza *FETCH* instrucțiune a celei care urmează). Odată sesizată întreruperea INT de către procesor acesta își va termina instrucțiunea în curs de execuție după care, dacă nu există activată o cerere de întrerupere sau de magistrală mai prioritară, nu va face faza *FETCH* a următoarei instrucțiuni ci va trece la pasul 2.
2. Identificarea întreruperii: procesorul va inițializa așa 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 - ieșire. 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. Informația de la locația specificată reprezintă adresa primei instrucțiuni din rutina de tratare a întreruperii, valoare care se încarcă în registrul PC. Vectorilor de întrerupere diferiți îi vor corespunde adrese de rutine diferite.

3. Procesorul va salva pe stivă PC-ul aferent instrucțiunii imediat următoare instrucțiunii executate de către procesor din programul principal (adresa de revenire), pentru a putea ști la finele rutinei de tratare a întreruperii unde să revină exact în programul principal. De asemenea, sunt salvate pe stivă codurile de condiție (N, Z, P) – vezi exemplul din figurile 9.7÷9.10. Acest fapt se poate dovedi insuficient având în vedere că în cadrul rutinei de serviciu pot fi alterați anumiți regiștri interni ai procesorului – alterare care poate fi chiar dezastruoasă la revenirea în programul principal. Din acest motiv cade în sarcina celui care scrie rutina de serviciu să salveze (folosind accese de depunere în stivă – PUSH) respectiv să returneze corespunzător (folosind accese de extragere din stivă – POP) acești regiștri.
4. După încheierea rutinei de tratare a întreruperii, instrucțiunea RTI restaurează codurile de condiție și vechiul PC (adresa de revenire). Este necesară practic o altă instrucțiune de revenire deoarece instrucțiunea RET preia doar adresa de revenire din registrul R7 și o depune în PC, nu și codurile de condiție.

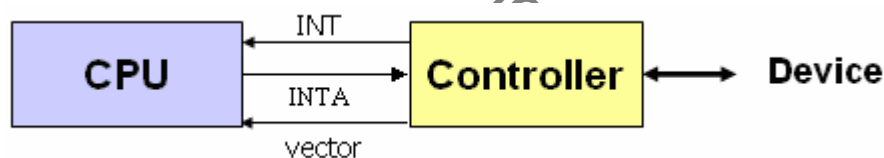


Figura 9.5. Modul de lucru prin întreruperi hardware.

În cazul în care mai mult de un dispozitiv extern dorește să întrerupă activitatea procesorului în același 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 activității procesorului prin apăsarea unei taste de către utilizator. La majoritatea dispozitivelor de intrare / ieșire registrul de stare conține pe lângă bitul *Ready* (care marchează disponibilitatea perifericului în generarea unei întreruperi) și 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: $INT = IE \text{ AND Ready}$. 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 neîntreruptabilă, 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.).

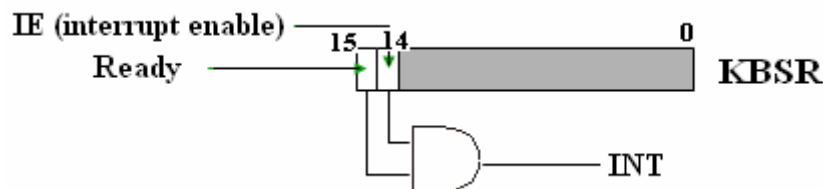


Figura 9.6. Generarea cererii perifericului de întrerupere a activității procesorului

În continuare sunt prezentate o secvență de ipostaze care justifică **necesitatea stivei în cazul tratării întreruperilor hardware imbricate**. În figura 9.7 programul principal A execută instrucțiunea de adunare ADD de la adresa 0x3006, moment în care este întrerupt de către perifericul B.

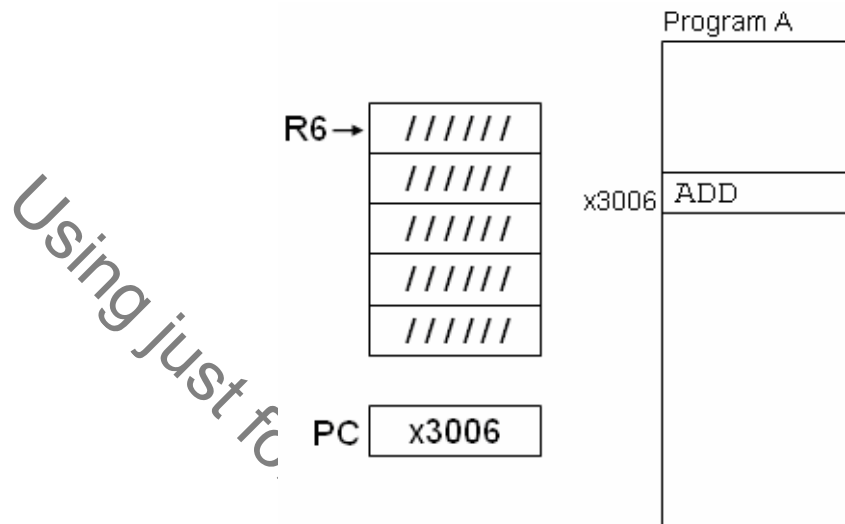


Figura 9.7. Programul principal A execută instrucțiunea ADD de la adresa 0x3006 când perifericul B îl întrerupe

La apariția semnalului INT, procesorul încheie instrucțiunea de adunare ADD, salvează pe stivă adresa următoarei instrucțiuni din programul principal (PC=0x3007) și codurile de condiție tocmai setate de rezultatul adunării și încarcă în PC valoarea vectorului de întrerupere, citește din memorie de la adresa respectivă adresa primei instrucțiuni din rutina de tratare (0x6200) pe care o încarcă în PC, predând controlul rutinei de serviciu.

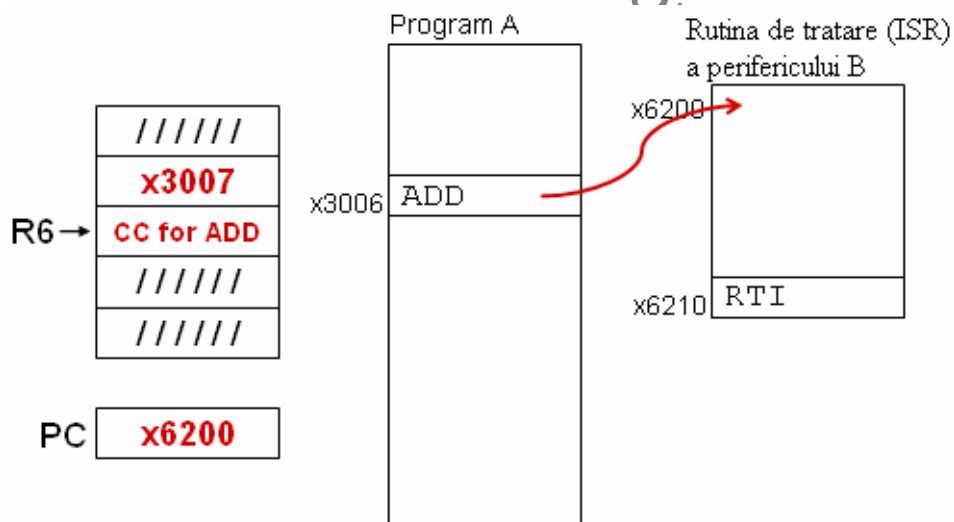


Figura 9.8. Protocolul de achitare a întreruperii – pașii 2 și 3

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.

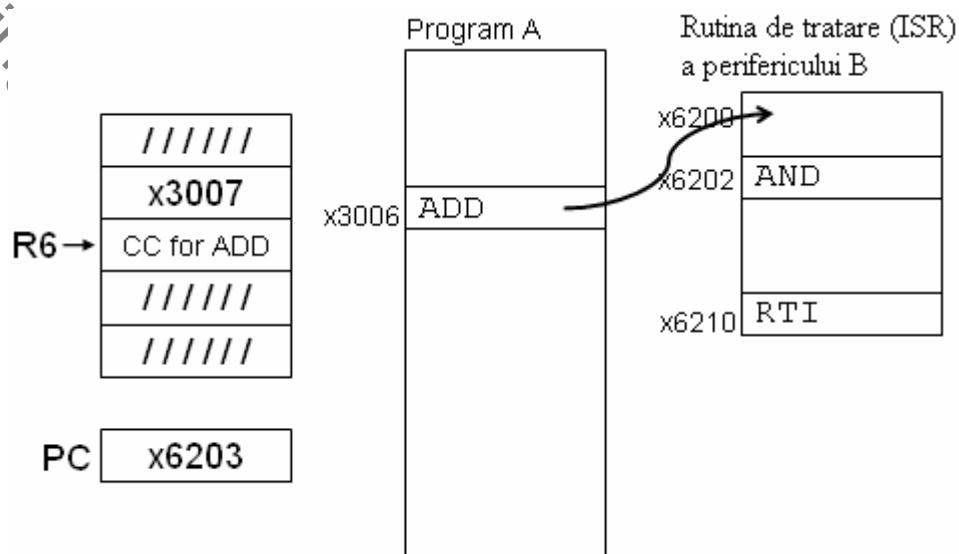


Figura 9.9. Instantaneu cu execuția instrucțiunii AND din rutina de serviciu a perifericului B

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`) și codurile de condiție (registrii booleeni N, Z și P). Rutina de serviciu aferentă perifericului C este plasată la adresa 0x6300 și conține 22 de instrucțiuni.

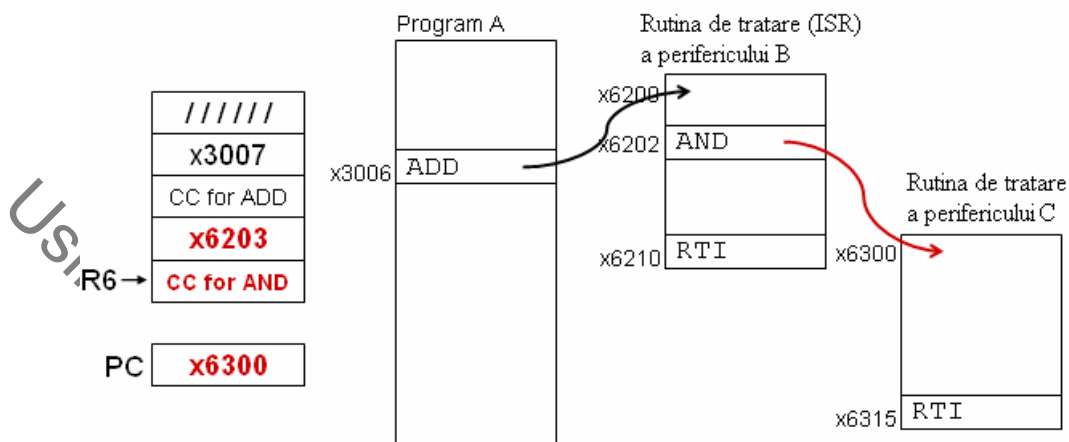


Figura 9.10. Instantaneu cu stiva, programul principal și cele două rutine de serviciu

Execuția instrucțiunii RTI (*return from interrupt*) 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 și 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.

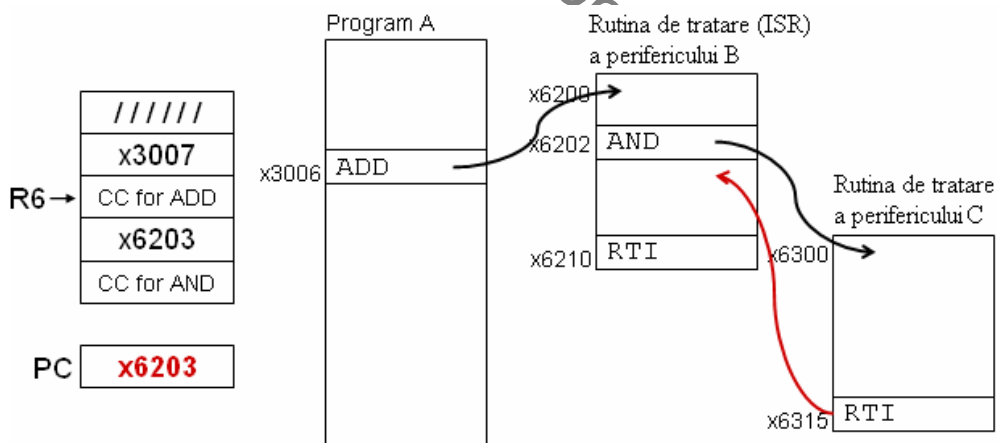


Figura 9.11. Revenirea din rutina de serviciu aferentă perifericului C

Execuția instrucțiunii RTI (*return from interrupt*) 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 și a adresei

instrucțiunii succesoare ADD-ului pe care a apărut întreruperea de la perifericul 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 perifericul B) când stiva era liberă.

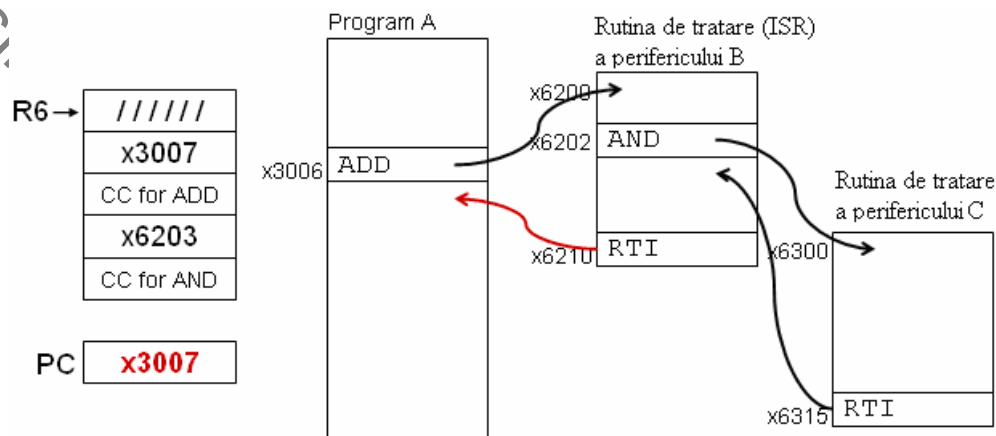


Figura 9.12. Efectul instrucțiunii RTI asupra stivei

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 perifericul B lansează o cerere de întrerupere. La finele fazei de *Sciere 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 perifericul 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 *Sciere 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 perifericul 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 instrucțiunile din rutina de serviciu aferentă perifericului B. Execuția instrucțiunii RTI de la adresa 0x6210, ultima din rutina de tratare aferentă perifericului B, presupune extragerea din stivă atât a codurilor de condiție cât și a adresei instrucțiunii succesoare ADD-ului pe care a apărut întreruperea de la perifericul B, marcând practic etapa cu numărul 4. În continuare sunt executate instrucțiunile din programul principal ca și cum nimic nu s-ar fi întâmplat, cu o oarecare întârziere din punct de vedere al timpului de execuție al programului A. În acest moment se observă cum indicatorul de vârf al stivei revine pe poziția inițială (dinaintea apariției întreruperii provocate de perifericul B) când stiva era liberă. Totuși anumite locații în stivă au fost afectate, chiar dacă acum ele figurează ca libere.

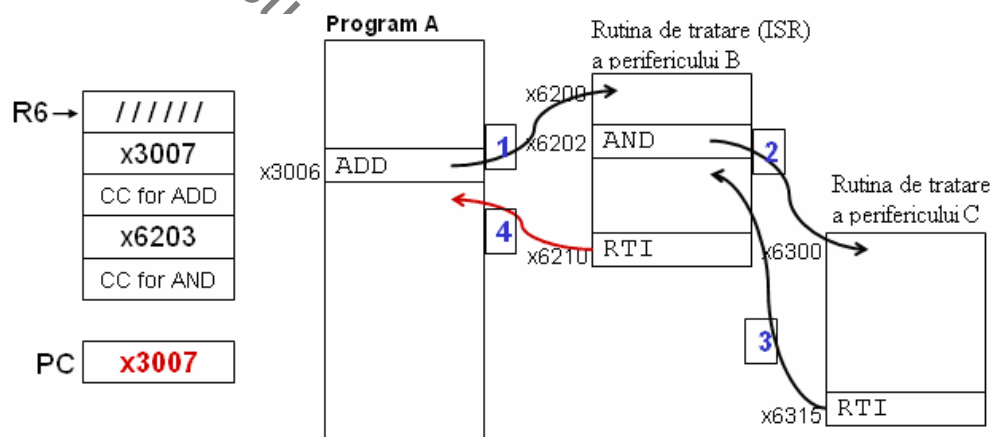


Figura 9.13. Instantaneu care sintetizează etapele parcurse la apariția întreruperilor de la cele două periferice

9.2. OPERAȚII ARITMETICE FOLOSIND STIVA

O aplicabilitate importantă a mecanismului stivei o reprezintă implementarea mașinilor virtuale. De regulă, aceste arhitecturi folosesc în locul regiștrilor o stivă pentru operanzi sursă și destinație (se mai numesc arhitecturi fără operanzi – „zero-address machine”). Instrucțiunile în acest caz sunt simple: *ADD*, *MUL*. Execuția lor presupune extragerea din stivă a ultimelor două locații ocupate, execuția propriu-zisă a adunării sau înmulțirii

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 declanșarea operațiilor care au loc la nivelul mașinii virtuale.

Pentru exemplificare se consideră următoarea expresie de calculat: $E = (A+B) \cdot (C+D)$. Dacă A reține valoarea 25, B conține valoarea 17, C este egal cu 3 și 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 $E = (A+B) \cdot (C+D)$ 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 **mașina virtuală Java** (JVM). Aceasta reprezintă un model abstract de arhitectură de calcul destinată execuției unui program scris într-un limbaj independent de mașină (bytecode Java).

- ✓ Interacționează atât cu sistemul de operare cât și 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 mașinii virtuale Java:

- *Registri* – include un program counter (*PC*) și trei alți regiștri folosiți pentru a administra *stiva*. Întrucât JVM este o arhitectură bazată pe *stivă* rezultă necesitatea unui număr redus de regiștri.
- 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 și rezultatele executării instrucțiunilor *bytecod* (rezultatele returnate de metode) sunt stocate pe *stivă*.
- *Stiva* este folosită și pentru a menține contextul fiecărei metode invocate, numit *stack frame*. Pentru a administra *stack frame*-ul (contextul), se folosesc regiștrii *optop*, *frame* și *vars*. Parametrii instrucțiunilor *bytecod* precum și rezultatele execuției acestor instrucțiuni sunt memorate în *stivă*.

O altă aplicabilitate importantă a mecanismului stivei se regăsește 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

Notăția poloneză (sau *postfixată*) este un mod de scriere a expresiilor, în care ordinea operatorilor și a operanzilor este schimbată față de cea dintr-o expresie uzuală.

În notația uzuală (numită și *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. Dacă întâlnim un operator atunci se extrag două valori din vârful stivei, care sunt, de fapt, cei doi operanzi, se aplică operatorul asupra valorilor extrase și rezultatul este depus în stivă. În momentul în care s-a terminat parcurgerea expresiei, rezultatul final al expresiei este în vârful stivei (și 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 fișiere) preiau datele în format ASCII (caractere, nu valori numerice). De asemenea, ieșirea informației din sistem se face în același 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	; afișează caracterul al cărui cod ASCII se află în R0
TRAP 0x25	; HALT – întrerupere care marchează încheierea programului

Dacă utilizatorul introduce succesiv valorile 2 și apoi 3 rezultatul afișat pe ecran va fi caracterul „e”. Explicația 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 citește de la tastatură numere alcătuite dintr-o cifră. În continuare va fi prezentată o secvență de program care permite conversia unui șir de caractere într-un număr alcătuit din mai mulți digiți. Se consideră astfel că, de la tastatură se citește șirul de caractere "259" și se stochează într-o zonă de memorie, de unde va fi luat fiecare caracter în parte, transformat în digit și prelucrat pentru generarea numărului final. Etapele conversiei sunt următoarele:

- ⌚ Se convertește primul caracter al șirului (cel mai semnificativ – în acest caz 2) în digitul corespunzător (se scade valoarea 0x30) și se înmulțește cu 100.
- ⌚ Se convertește al doilea caracter al șirului (în acest caz 5) în digitul corespunzător și se înmulțește cu 10.
- ⌚ Se convertește al treilea caracter al șirului (în acest caz 9) în digitul corespunzător. Dacă numărul ar fi avut mai mulți digiți trebuia inițial identificată lungimea șirului, iar aflarea fiecărei cifre continua până la cifra unităților.
- ⌚ Se adaugă toate cele trei numere obținute la fiecare pas.

Una din operațiile amintite anterior și necesară în procesul de conversie o reprezintă înmulțirea. Întrucât arhitectura LC-3 ISA nu dispune de o instrucțiune de înmulțire trebuie găsită o alternativă prin care să se poată realiza înmulțirea unui digit cu 100 și apoi înmulțirea unui digit cu 10. O abordare simplistă ar consta în adunări repetate (se adună digitul respectiv de 100 de ori), variantă inefficientă totuși. O a doua variantă o reprezintă adunarea numărului 100 cu el însuși de *valoarea digitului* respectiv ori (mai eficientă decât varianta anterioară întrucât se fac mai puține adunări – digitul fiind în mod clar <10). Chiar și această a doua variantă se poate

Rutina de conversie completă din ASCII în decimal

; Buffer-ul de trei digiți se numește ASCIIBUF.

; R1 indică numărul de digiți care trebuie converțiți. 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
		rămași pentru conversie
		; deși 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 pregătește
		decrementarea codului ASCII al
		caracterului cu inversul codului
		cifrei „0”
	LEA R2, ASCIIBUF	; R2=adresa de bază a primului
		element din șir (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	; citește din tabelă caracterul de
		convertit
	ADD R4, R4, R3	; convertește 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 converțiți,
		cu implicații implicite și 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	; citește din tabelă caracterul de convertit
	ADD R4, R4, R3	; convertește caracterul în digit
	LEA R5, Lookup10	; R5 reține adresa tabelii cu zeci (multiplii de 10)
	ADD R5, R5, R4	; se determină offsetul în tabela zecilor -> R5
	LDR R4, R5, #0	; în R4 se depune multiplul de 10 respectiv
	ADD R0, R0, R4	; se adaugă la unități și 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	; citește din tabelă caracterul de convertit
	ADD R4, R4, R3	; convertește caracterul în digit
	LEA R5, Lookup100	; R5 reține adresa tabelii cu sutele (multiplii de 100);
	ADD R5, R5, R4	; se determină offsetul 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 și 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 șirul de caractere de trei digiți și pentru terminatorul NULL
Lookup10	.FILL 0	

9.3.2. CONVERSIA DIN ZECIMAL ÎN ASCII

Primul pas îl reprezintă verificarea semnului numărului. Semnul este scris în buffer urmând ca, dacă numărul este negativ să fie transformat într-unul pozitiv.

; R0 reprezintă un număr întreg de trei cifre din intervalul -999 și +999.
; În buffer-ul ASCIIBUF se înscrie pe prima poziție semnul urmat de trei
digiti în format ASCII.

LEA R1, ASCIIBUF	; R1 – adresa șirului de caractere care reține rezultatul
ADD R0, R0, #0	; se testează semnul numărului
BRn NegSign	; dacă este număr negativ se sare la secvența care scrie semnul în

		buffer și 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 și 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 obții 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 și 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 și 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 și mai scad 10
End10 STR R2, R1, #2 ; memorez digitul zecilor în buffer
                    ; pe poziția a treia – după semn și
                    ; 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 și 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 0xFFFF6 ; inversul valorii 10 (pentru scăderi
                    ; repetate ale zecilor)

```

9.4. EXERCITII ȘI PROBLEME

1. Care sunt caracteristicile definatorii ale stivei? Care este rolul indicatorului spre vârful stivei (*stack pointer*)? Ce operații cu stiva cunoașteți și cum sunt ele implementate din punct de vedere al accesului la memorie? Există anumite condiții (cazuri speciale) care trebuie tratate?

2. Se consideră un șir 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

șirul de intrare este următorul: ABCDEFGHIJKLM. Șirul de ieșire reprezintă totalitatea elementelor care se extrag din stivă în ordinea stabilită de program (programator).

- a) Care este șirul de ieșire pentru șirul de intrare specificat anterior? Indicație: BD...

- b) Dacă șirul de intrare este ZYXWVUTSR, creați o secvență de operații cu stiva (push și respectiv pop) astfel încât șirul de ieșire să fie YXVUWZSRT.

3. Se reamintește 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 și o depune în memorie la adresa M.
- **OP** – Extrage primele două valori aflate în vârful stivei și execută operația aritmetico-logică specificată asupra celor două valori. Rezultatul este încărcat înapoi pe stivă.

Pentru această problemă opcode-urile (OP) necesare sunt: ADD și 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 și instrucțiunea de înmulțire – MUL.

- b) Specificați câte un avantaj și respectiv un dezavantaj al fiecărui tip de mașină din cele anterior descrise.

4. Se consideră cele două operații de depunere în stivă – Push și 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 și o depune în registrul specificat Rn . Mai jos este descris un instantaneu cu cei opt regiștrii ai arhitecturii LC-3 înainte și după execuția a șase operații cu stiva. Trebuie menționat că patru din șase operații nu sunt complet specificate. Completați în locul celor patru linii goale cu numele regiștrilor 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

- Care este conținutul stivei după ultima depunere – **PUSH F**?
- În care moment de timp stiva conține numărul maxim de elemente?
- 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:

- Care sunt cele mai uzuale operații cu stiva și care este modul lor de lucru ? Restricții ce pot să apară.
- 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 și 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 și 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 și 0 în caz contrar.

Atât funcțiile *Pop* cât și *Top* necesită testarea condiției ca stiva primită ca parametru să nu fie vidă.

8. Implementați o mașină virtuală bazată pe stivă (*zero-address machine*), folosind ca și arhitectură de bază (gazdă) LC-3 ISA: instrucțiuni,

registrii – cu nume și funcțiile stabilite (R6 – *stack pointer*, etc.). Instrucțiunile mașinii virtuale vor fi: *OpAdd*, *OpMult*, *OpNeg*, *Push M*, *Pop M*. Implementați toate aceste instrucțiuni și rulați programele din subcapitolul 9.2 și evaluați timpul de execuție (eventual număr de instrucțiuni aferente arhitecturii gazdă executate) în ambele situații.

9. Implementați într-un program de nivel înalt (C, C++, Java) următoarea cerință: Se citește dintr-un fișier o expresie aritmetică în care operanzii sunt simbolizați prin litere mici (de la „a” la „z”) iar operatorii sunt „+”, „-”, „*”, „/” cu semnificația cunoscută. Se pot folosi și parantezele rotunde: „(” și „)”. 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 ține cont de următoarele indicații: o expresie (S) este alcătuită din termeni (T), separați prin „+” sau „-”; un termen este alcătuit din factori (F) separați prin „*” sau „/”; un factor este sau o literă mică sau o expresie cuprinsă între paranteze.
10. Implementați într-un program de nivel înalt (C, C++, Java) următoarea problemă: Se citește dintr-un fișier o expresie aritmetică în care operanzii sunt simbolizați prin litere mici (de la „a” la „z”) iar operatorii sunt „+”, „-”, „*”, „/”, „(” și „)” cu semnificația cunoscută. Se cere să se convertească această expresie într-o expresie în forma poloneză (postfixată) – vezi subcapitolul 9.2.
11. Implementați într-un program de nivel înalt (C, C++, Java) următoarea problemă: Dându-se o expresie în forma poloneză (postfixată), realizați programul care calculează expresia utilizând numai instrucțiuni de atribuire cu un singur operator. Se vor introduce variabilele auxiliare x_1, x_2, \dots, x_n , unde n reprezintă numărul operațiilor care apar în expresie. Să se verifice aplicația pentru diverse expresii și pentru diverse valori aferente fiecărui operand (simbol).

10. FACILITĂȚI ALE MAȘINII PENTRU IMPLEMENTAREA ÎN HARDWARE A FUNCȚIILOR DIN PROGRAMELE DE NIVEL ÎNALT. STIVA DE DATE AFERENTĂ FUNCȚIILOR

10.1. OBTINEREA CODULUI OBIECT PENTRU O ARHITECTURĂ DATĂ

Limbajele de programare de nivel înalt sunt independente de mașina (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 (registrii, 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, și pentru [portabilitatea](#) sa. A fost dezvoltat la începutul anilor 1970 de Brian Kernighan și Dennis Ritchie, care aveau nevoie de un limbaj simplu și portabil pentru scrierea [nucleului](#) sistemului de operare UNIX. Sintaxa limbajului C a stat la baza multor limbaje create ulterior și 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 aceeași problemă [Patt03]. De asemenea, LC-3 nu asigură instrucțiuni de înmulțire și 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.

și expresii simple pentru structuri de control (if-else, switch-case) sau repetitive (for, while, do-while). Permite compilarea condiționată. Deși 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 și un dezavantaj, în special pentru programatorii fără experiență.

Compilare vs. Interpretare

Programele de nivel înalt (*High Level Languages* – HLL) pot fi traduse în instrucțiuni mașină (ISA corespunzătoare arhitecturii hardware pe care se va procesa) prin două metode [Aho86, Gol97]:

✓ **Interpretare:**

Codul sursă HLL este „interpretat” de o mașină virtuală (cele mai cunoscute sunt mașinile virtuale Java – JVM) care translatează „secțiuni” din codul sursă în limbaj mașină și 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 și faptul că nu necesită o zonă mare de memorie pentru stocarea programului compilat. Principalul dezavantaj îl reprezintă viteza scăzută de execuție a aplicației, dar și necesitatea existenței în momentul interpretării atât a codului sursă HLL cât și a mașinii virtuale care realizează interpretarea.

✓ **Compilare:**

Compilarea directă translatează codul sursă (poate fi Java, C, C++, Fortran, Pascal) în instrucțiuni mașină, 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 și 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 același 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* aparținând unui compilator se înțelege o citire a programului dintr-un fișier, transformarea lui conform unor faze și scrierea rezultatului în alt fișier(de ieșire).

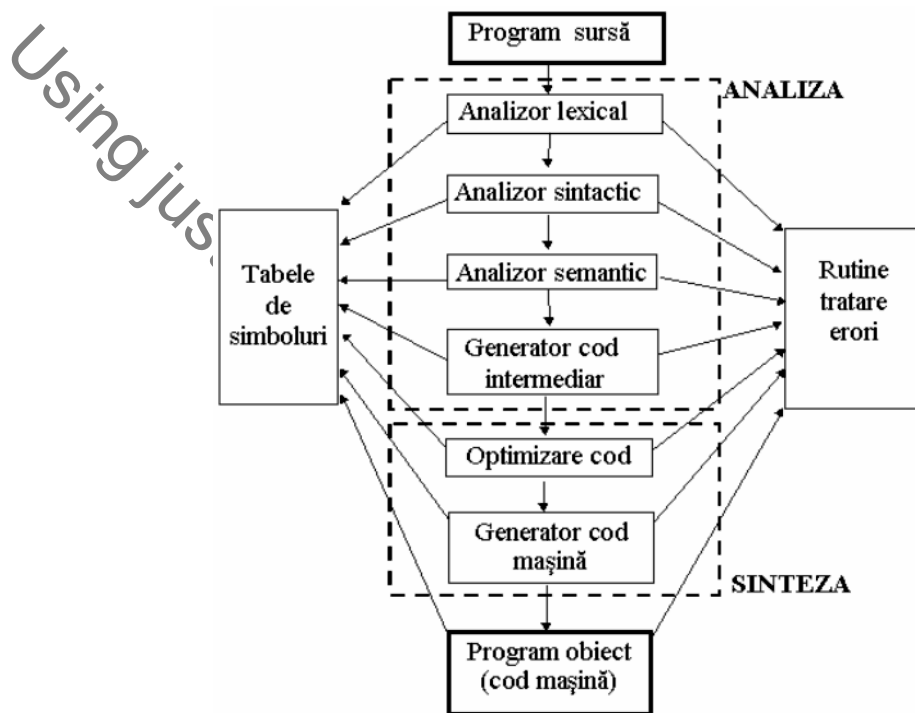


Figura 10.1. Fazele procesului de compilare

Figura anterioară (10.1) ilustrează cele două faze majore ale procesului de compilare propriu-zisă:

- ☞ **analiza** (de tip “*front end*”) - în care se identifică părțile constitutive fundamentale ale programului (variabile, expresii, declarații, definiții, apeluri de funcții) și se construiește o reprezentare internă a programului original, numită „*cod intermediar*” (*analiza lexicală* produce un -> șir de atomi lexicali -> *analiza sintactică* generează -> arborele sintactic -> *analiză semantică* construiește o reprezentare a programului sursă în-> *cod intermediar*). Este dependentă de limbajul de nivel înalt și nu de mașina pe care se va procesa.
- ☞ **sinteza** (de tip “*back end*”) - generează cod mașină eventual optimizat. Se disting două etape:
 - optimizare cod intermediar (scheduling) pentru o anumită mașină;
 - generare de cod mașină (generare cod într-o gamă variată de formate: *limbaj mașină absolut*, *limbaj mașină relocabil* sau

limbaj de asamblare urmat de alocare de resurse). Această fază este puternic dependentă de mașina pe care se va executa codul obiect generat.

Înainte de prima fază 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 destinație (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 conținutul fișierului *<stdio.h>* în codul sursă al aplicației create acolo unde apare respectiva directivă de preprocesare, sau, să înlocuiască anumite valori constante declarate ca șiruri 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, și care este responsabilă de transformarea programului sursă văzut ca o succesiune de caractere (text) într-o succesiune de atomi lexicali și de atribute ale lor. Conform definiției din [Gol97], un atom lexical este o entitate indivizibilă a unui program - identificatori de variabile/funcții, 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 considerente care demonstrează utilitatea analizei lexicale ca fază separată a compilatorului:

1. Asigură o proiectare mai ușoară a compilatorului.
2. Faza necesită operații de nivel fizic: citiri de fișiere disc (programul sursă), comparări de șiruri de caractere, acces la nivel de bit, căutări 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 și semantică este simplificat textul fiind curățat de spații (blank-uri) și comentarii.
4. Asigurarea portabilității algoritmului de compilare pentru diverse versiuni ale aceluiași limbaj de programare.

Analizorul lexical (AL) constituie interfața dintre programul sursă și compilator (analizorul sintactic). Relația analizorului lexical (AL) față de analizorul sintactic (AS - cel care determină faptul că atomii lexicali aparțin sau nu limbajului de programare) este de două feluri:

- a) **independent** - AL este independent de AS. Analizorul lexical primește la intrare textul cu programul sursă și furnizează la ieșire un fișier sau o zonă de memorie cu succesiunea de atomi lexicali ce vor fi dați spre prelucrare analizorului sintactic (vezi 10.2).

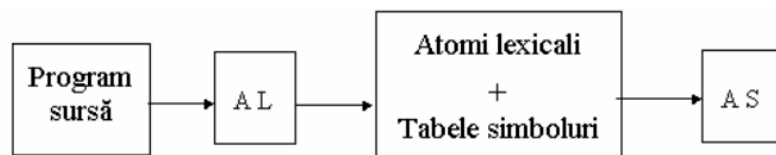


Figura 10.2. Analizorul Lexical independent de Analizorul Sintactic (1)

- 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 și AL sunt corutine (două procese independente care se lansează reciproc - în sisteme multitasking UNIX, Windows).

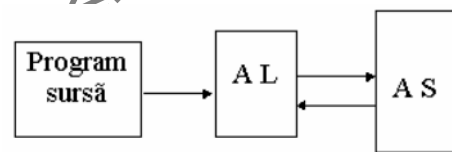


Figura 10.3. Analizorul Lexical comandat de Analizorul Sintactic

Analiza sintactică grupează atomii lexicali generați de AL în structuri sintactice precum: expresii, liste, instrucțiuni, proceduri. Toate acestea le plasează într-un *arbore sintactic*-cu ajutorul căruia sunt descrise relațiile de incluziune ale structurilor sintactice, furnizând în final informații privitoare la corectitudinea acestora.

Analiza semantică - (componenta compilatoarelor cel mai puțin dezvoltată teoretic), face delimitările din punct de vedere semantic (al înțelesului programului). Extrage informațiile privind aparițiile obiectelor purtătoare de date din program (variabile, tipuri de date, funcții, etc) pentru verificarea consistenței utilizării lor. Pe măsura parcurgerii arborelui, analiza semantică construiește o reprezentare a programului sursă în **cod intermediar** (șir de instrucțiuni simple cu format fix, foloseau operanți variabile de program și nu regiștri), reflectând structura programului sursă și înglobând toate informațiile necesare fazelor următoare : *optimizarea* și *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 lungeste corespunzător durata

compilării. Numărul acestor treceri poate fi fix ori, în cazul unor limbaje mai complexe sau a celor defectuos proiectate, variabil în funcție de structura programului analizat.

Toate fazele unui compilator dispun de **rutine de tratare a erorilor** și lucrează în comun cu una sau mai multe tabele de simboluri în care se păstrează informații despre cuvintele cheie ale limbajului, identificatorii de variabile (tip, lungime, adresă stocare în memorie etc.), etichete de instrucțiuni, identificatori de proceduri și funcții și alte informații utile despre programul în curs de compilare. **Tabela de simboluri** este creată în faza de analiză a programului sursă și folosită în scopul validării numelor simbolice facilitând generarea de cod [Gol97]. Tabela de simboluri realizează o asociere simbolică între numele (identificatorul) unei variabile și caracteristicile acesteia (tip, domeniu de vizibilitate, deplasament față de începutul zonei de date statice sau dinamice, stivă). Compilatorul folosește tabela de simboluri pentru a urmări domeniul de utilizare și valabilitate a unui nume și de a adăuga informații despre un nume. Tabela de simboluri este cercetată de fiecare dată când un nume este întâlnit în textul sursă. Mecanismul asociat tablei de simboluri trebuie să permită adăugarea a noi intrări și găsirea informațiilor existente în mod eficient. Tabela de simboluri este similară celei generate de asamblor însă cuprinde mai multe informații decât aceasta. În asamblare, toți identificatorii (variabilele) erau etichete (*labels*) și informațiile reprezentate de acestea erau adrese. Tabela de simboluri conține informații legate de toate variabilele din program.

Optimizarea de cod intermediar (**scheduling**) [Vin00, Gol97], deși opțională, este extrem de importantă din punct de vedere al creșterii performanței în domeniul paralelismului la nivelul instrucțiunilor. Optimizarea poate fi tratată din două perspective: **locală** (în cadrul unităților secvențiale de program – basic-block-uri) și respectiv **globală** (a întregului program). Noțiunea de *optimizare* este oarecum forțată întrucât nu există garanția 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 și pentru mașini secvențiale cu un număr limitat de registre în vederea stocării temporare a variabilelor, este de așteptat ca gradul de dependențe între instrucțiunile adiacente să fie ridicat (paralelism relativ scăzut (2-3 instrucțiuni) la nivelul basic-block-urilor). Așadar pentru creșterea nivelului de paralelism este necesară suprapunerea execuției unor instrucțiuni 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, și care la ora actuală constituie încă o problemă deschisă, se bazează pe algoritmi determinați de tip "*Trace Scheduling*" sau pe algoritmi euristici de tip "*Enhanced Percolation*", "*inlining*" selectiv aplicat procedurilor [Vin00].

Dependențele cauzate de variabilele aflate în memorie reprezintă de asemenea o frână în calea creșterii performanței arhitecturilor de calcul. Procesarea «*Out of Order*» a instrucțiunilor cu referire la memorie într-un program este dificilă datorită accesării aceleiași adrese de memorie de către o instrucțiune care citește respectiv una care scrie. Există motive însă, ca o instrucțiune *Load* amplasată după o instrucțiune *Store* să se execute înaintea acesteia din motive de eficiență a execuției (mascare latență, 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 secvenței se modifică inacceptabil. Atunci când acest lucru este posibil, problema se rezolvă static, de către compilator. Rutina de **analiză anti-alias** (*disambiguation routine*) [Vin00], componentă a compilatorului, compară cele 2 adrese de memorie și returnează una dintre următoarele 3 posibilități:

- adrese întotdeauna distincte;
- adrese întotdeauna identice;
- cel puțin 2 adrese identice sau nu se poate determina.

Doar în primul caz putem fi siguri că execuția anterioară a instrucțiunii *Load* față de instrucțiunea *Store* (sau simultană în cazul procesoarelor cu execuție multiplă) îmbunătățește performanța 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 și 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 acțiunile 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 și implicit costuri sporite.

Pe lângă dezambiguizarea statică mai există și alte transformări de cod în vederea optimizării [Gol97]:

- Eliminarea instrucțiunilor redundante sau a celor de neatins (*unreachable code*) din program. De exemplu, o instrucțiune neetichetată care urmează imediat după un salt necondiționat poate fi eliminată.

- Optimizări ale fluxului de control. De exemplu, cazul instrucțiunilor de salt condiționate sau nu, care au ca destinație tot o instrucțiune de salt, sau salt condiționat.
- Simplificări algebrice.
- Reduceri în forță. De exemplu, înmulțirea sau împărțirea întregilor (în virgulă fixă) printr-o putere a lui 2 este mai ieftin de implementat ca o deplasare (shift) de biți. Împărțirea în virgulă flotantă printr-o constantă poate fi implementată (aproximată) ca și multiplicare a constantei, ceea ce poate fi ieftin.

Generatorul de cod constituie faza finală a unui compilator. Primește la intrare reprezentarea intermediară a programului sursă împreună cu informația din tabela de simboluri - folosită la determinarea run-time a adresei obiectelor de date, desemnate de nume în reprezentarea intermediară, și produce la ieșire un program obiect echivalent. Codul de ieșire trebuie să fie corect și de înaltă calitate, aceasta însemnând că el trebuie să folosească eficient resursele mașinii pentru a putea fi procesat cât mai rapid.

Ieșirile generatorului de cod se găsesc într-o gamă variată de forme: *limbaj de asamblare*, *limbaj mașină relocabil* sau *limbaj mașină absolut*. Producând un program într-un limbaj de asamblare, caracterizat de o mai mare lizibilitate, generarea codului se realizează mult mai ușor. Pot fi generate instrucțiuni simbolice precum și utilizate facilitățile de macro-asamblare ajutătoare la generarea de cod. Producând un program în limbaj mașină absolut, există avantajul că programul poate fi plasat într-o locație fixă a memoriei și poate fi imediat executat. Producând un program în limbaj mașină relocabil, acesta permite subprogramelor să fie compilate separat (similar cu fișierul *Makefile* din cadrul setului de instrumente SimpleScalar 3.0 [Flo05]). O mulțime de module-obiect relocate pot fi *legate (linked)* împreună și apoi încărcate pentru execuție de către un *loader*. Câștigul de flexibilitate care dă posibilitatea compilării separate a subrutinelor și apelarea altor programe compilate anterior dintr-un modul obiect este tribut ar costurilor suplimentare pentru legare și încărcare.

Dintre problemele inerente care apar la generatoarele de cod se amintesc: managementul de memorie (mai ales în cazul instrucțiunilor de salt condiționat), selecția instrucțiunilor mașină, alocarea registrelor și ordinea de evaluare.

Natura setului de instrucțiuni al mașinii destinație (viteza instrucțiunilor și limbajul mașinii) determină dificultatea în selectarea instrucțiunilor ce vor fi folosite pentru generarea de cod obiect. Uniformitatea și completitudinea setului de instrucțiuni constituie de asemenea, factori importanți. Dacă mașina destinație nu suportă toate

tipurile de date într-o manieră uniformă, atunci fiecare excepție are nevoie de o tratare deosebită.

Este cunoscut faptul că instrucțiunile cu operanzii registru (la nivelul procesorului) sunt de obicei mai scurte și 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 mulțimea de variabile care va fi rezidentă în regiștrii.
2. Din acel punct, în timpul următoarei faze de asignare a regiștrilor (la variabile) se va selecta registrul specific ce conține variabila rezidentă în el.

Găsirea unei asignări optime a regiștrilor la variabile este dificilă, hardware-ul și software-ul mașinii destinație putând solicita anumite convenții în utilizarea anumitor regiștrii - folosirea specializată a unora dintre aceștia. De exemplu, registrul \$29 reprezintă *stack pointer*-ul, \$31 - constituie adresa de revenire, sau regiștrii flotați dublă precizie sunt alcătuiți din regiștrii par:impar flotați simplă precizie, în cazul procesorului MIPS.

De asemenea, ordinea în care sunt efectuate calculele poate afecta eficiența codului generat, unele calcule necesitând mai puține registre pentru rezultate intermediare, față de altele. Cu toate aceste probleme, compilatoarele sunt astfel proiectate încât să respecte principiile de calitate (corectitudine) și eficiență.

În finalul acestui paragraf este exemplificat modul de alocare al spațiului 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:

- Instrucțiunile (zona de *cod*) la care registrul PC (*program counter*) indică spre adresa următoarei instrucțiuni 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 și sub numele de zonă de date statice. Această zonă se caracterizează prin faptul că, în momentul compilării, se cunoaște numele, adresa și spațiul ocupat de fiecare variabilă declarată aici (static).
- Stiva de date (aferentă fiecărei funcții din programul sursă HLL). Aici sunt stocate variabilele locale, parametrii funcțiilor. R6 (similar cu

registru \$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 ieșirii 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 și 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 și ceilalți parametri (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

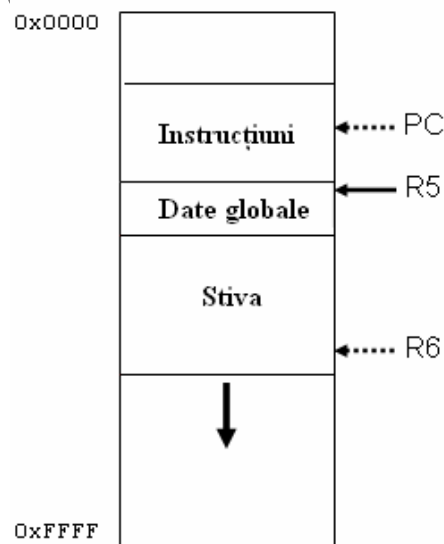


Figura 10.4. Organizarea memoriei la calculatorul LC-3

10.1.2. ETAPELE PARCURSE PENTRU RECOMPILAREA INSTRUMENTELOR *SIMPLESCALAR 3.0* [Flo05]

Creșterea în performanță și complexitate a microprocesoarelor moderne datorate tehnicilor avansate gen *pipelining*, execuție *out-of-order*, predicție și execuție speculativă, presupune un efort suplimentar de

proiectare și verificare pentru dezvoltarea și implementarea de produse viabile. Pentru depășirea acestor probleme, proiectanții de microarhitecturi au explorat diverse modalități de *transfer de funcționalitate* la nivelul compilatorului. Începând cu procesoarele RISC¹⁵ VLIW¹⁶ și continuând cu cele EPIC¹⁷ – versiunile 1 și 2 de procesoare Intel Itanium, compilatorul a jucat un rol important în simplificarea arhitecturii la nivel hardware menținând totodată tendințele curente de creștere a performanței. În acest paragraf sunt prezentate fazele principale care trebuie parcurse pentru compilarea și execuția 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 și până la simularea de tip *execution driven* sunt evidențiate în figura următoare.

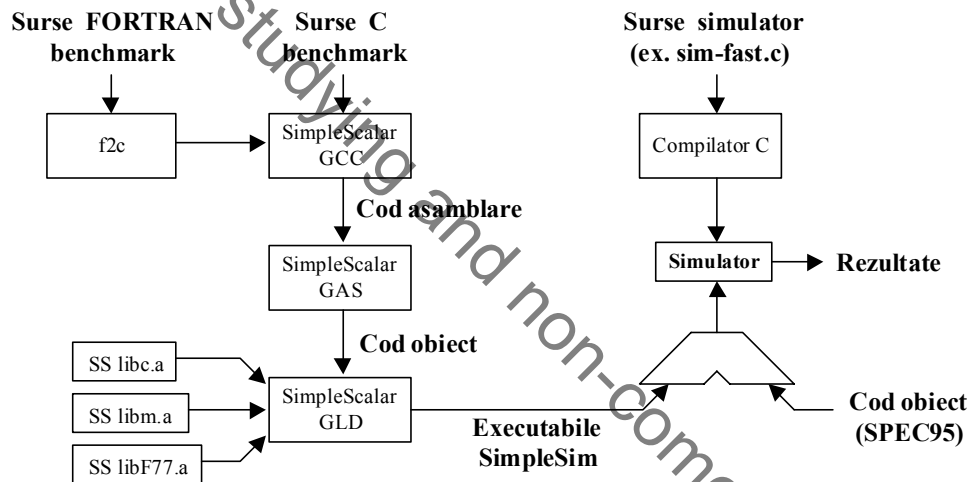


Figura 10.5. Interacțiunea instrumentelor în cadrul setului SimpleScalar

¹⁵ RISC – calculatoare cu set redus (optimizat) de instrucțiuni

¹⁶ VLIW – procesoare cu execuție multiplă (Very Long Instruction Word). Din punct de vedere hardware un procesor VLIW este foarte simplu și constă dintr-o colecție de unități funcționale de execuție (sumatoare, circuite multiplicative, unități de execuție branch-urilor, etc) conectate printr-o magistrală, plus regiștrii și memorii cache. În cazul procesoarelor VLIW compilatorul are un rol decisiv în gruparea mai multor instrucțiuni independente într-o singură instrucțiune multiplă astfel încât să poată fi folosite eficient toate unitățile de execuție pe perioada fiecărui impuls de tact.

¹⁷ EPIC – calculatoare care exploatează paralelismul în mod explicit. Procesorul nu trebuie să asigure paralelismul instrucțiunilor. Compilatorul este responsabil pentru determinarea instrucțiunilor care pot fi executate în paralel și codifică aceste informații direct în codul obiect obținut după compilare).

Sursele FORTRAN ale programelor de test proprii (*benchmark-uri*) sunt convertite în C folosind translatorul *f2c*. Atât *benchmark-urile* C cât și 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 convențională. Spațiul de adrese utilizator (pe 31 de biți) este compus din trei părți: cod, date și stiva program. Instrucțiunile sunt pe 64 de biți iar setul de regiștrii generali pentru numere întregi conține 32 de regiștrii pe 32 de biți 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 funcțiile de bibliotecă și directivele de asamblare). Formatul final de cod obiect se obține 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 colecție de compilatoare, asamblatoare, link-editoare, simulatoare), disponibil gratuit [Flo05], pune la dispoziție și câteva din *benchmark-urile* SPEC '95 în același format cod obiect (precompilate pentru arhitectura SimpleScalar).

Compilarea și Link-editarea programelor

La ora actuală există pe piață o varietate de compilatoare disponibile, atât pentru platformă Windows cât și 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 mașină specific arhitecturii SimpleScalar). Pe lângă compilator, furnizorul acestuia oferă și unelte suplimentare necesare: preprocesoare, asamblatoare și link-editoare. Compilatoarele oferă o mulțime de opțiuni de compilare, preprocesare, link-editare cu biblioteci (matematice, grafice, etc) și inclusiv facilități de depanare (*gdb* – *GNU debugger*). Pentru compilarea programelor se recomandă specificarea opțiunilor **-Wall** pentru afișarea tuturor avertismentelor generate de compilator. Prin opțiunea **-c** este evitată link-editarea iar **-g** invocă opțiunea 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

fișierelor sursă respective
fără a efectua legăturile cu
bibliotecile funcțiilor
apelate.

Link-editarea se poate realiza ulterior cu comanda:

```
gcc nume_fis1.o nume_fis2.o ..... -o nume_fis_executabil =>
determină codul mașină
(.exe pentru Intel, .ss
pentru SimpleScalar) care
poate fi executat pe
arhitectura gazdă (pentru
care s-a generat).
```

Se reamintește că opțiunea de compilare **-S** realizează preprocesarea și compilarea codului sursă, iar opțiunea **-s** generează doar codul asamblare fără faza de preprocesare. De asemenea, în cazul în care fișierele sursă **c** și **cpp** nu există în directorul unde se găsește compilatorul **gcc**, **xgcc**, etc atunci se va tasta odată cu numele și calea spre acest fișier.

```
./xgcc nume_fisier.c -S => fișier asamblare MIPS (s-a folosit xgcc)
./xgcc nume_fisier_sursa.c -o nume_fisier_destinatie.ss => fișier
executabil MIPS
```

10.2. IMPLEMENTAREA GESTIUNII STIVELOR DE DATE ASOCIATE FUNCȚIILOR C

10.2.1. SUBPROGRAME. GENERALITĂȚI. FUNCȚII C

Prin definiție un *subprogram* (ex. *funcție C*, *funcție* sau *procedură* în Pascal) reprezintă un ansamblu de date, variabile și instrucțiuni scrise în vederea unor prelucrări (calculare, citiri, scrieri) și 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 puțin o dată. Prin realizarea de subprograme se stabilesc premisele programării structurate conform ecuației programării structurate enunțată de Niklaus Wirth: „**Structuri de date + Algoritmi = Program**”. Caracterul generic (se operează asupra unei varietăți de date de

intrare) și concis (subprograme, structurile repetitive și 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 și chiar de același program de mai multe ori.
- Noțiunea de subprogram stă la baza mecanismului de recursivitate și a tehnicii de *divide et impera*.
- Facilitează elaborarea algoritmilor prin descompunerea problemei de rezolvat în altele mai simple, fluxul de execuție al programului putând fi urmărit cu o mai mare ușurință.
- Permite dezvoltarea modulară a codului (separat și independent) de către programatori independenți.
- Reducerea numărului de erori care pot apare la scrierea programelor.

FUNCȚII C [Zah04, Neg97, Sto98]

Funcția este un concept important în matematică și în programare. În limbajul C prelucrările sunt organizate ca o ierarhie de apeluri de funcții. Orice program trebuie să conțină cel puțin o funcție, funcția *main*. Funcțiile încapsulează prelucrări bine precizate și pot fi reutilizate în mai multe programe.

În esență, o funcție este alcătuită din:

- **Antet** – conține numele funcției, lista parametrilor formali, tipul rezultatului.

Tip_returnat nume_funcție (lista_parametrilor_formali)

Identice ca și în matematică o funcție nu poate întoarce mai mult de un rezultat. Există situații (efectuarea unor acțiuni, etc) când o funcție nu întoarce nimic (în aceste cazuri tip_returnat este *void*). Funcțiile care nu returnează nimic se numesc în alte limbaje de programare *proceduri*. Tipul returnat de funcție poate fi orice tip de dată cu excepția masivelor. Dacă se dorește ca o funcție să returneze un masiv atunci acesta trebuie înglobat în structuri de date declarate ca *struct*. Rezultatul funcției este returnat la întâlnirea instrucțiunii *return* din corpul funcției (*return* expresie). Trebuie ca tipul expresiei să coincidă cu tipul funcției. La întâlnirea instrucțiunii *return*, după atribuirea valorii, execuția funcției se încheie și se revine la funcția care a apelat-o. În absența instrucțiunii *return*, execuția funcției se încheie după execuția ultimei instrucțiuni. În acest caz funcția 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 Parametru₁, Tip_2 Parametru₂, ..., Tip_n Parametru_n

- **O instrucțiune compusă** – aceasta cuprinde declarațiile variabilelor locale și instrucțiunile propriu-zise.

Transmiterea parametrilor

Parametrii care se găsesc în antetul funcției se numesc formali (declarați în funcția apelată), iar cei care se găsesc în instrucțiunea de apel se numesc efectivi (*Arguments* – în engleză) și sunt declarați în (sub)programul apelant. Între parametrii formali și cei efectivi trebuie să existe o anumită concordanță, descrisă prin regulile următoare:

- Numărul parametrilor formali (și ordinea acestora) trebuie să coincidă cu numărul (și ordinea) parametrilor efectivi. Există și excepții însă: funcții cu număr variabil de parametri. În cazul acestora este obligatoriu ca cel puțin primul parametru să apară în lista parametrilor formali. Pentru parametrii variabili antetul funcției va conține „...” 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** și prin **referință**. Transmiterea parametrilor prin valoare se utilizează atunci când nu ne interesează ca, la întoarcerea din subprogram (funcție), parametrul efectiv să rețină valoarea modificată acolo (în subprogram). În cazul transmiterii prin valoare parametrii efectivi trebuie să fie valori reținute de variabile sau expresii. Transmiterea parametrilor prin referință se utilizează atunci când se dorește ca, la întoarcerea din subprogram, variabila transmisă să rețină valoarea stabilită în timpul execuției subprogramului. În cazul transmiterii prin referință parametrii efectivi trebuie să fie referințe la variabile. Dacă în cazul transmiterii parametrilor prin valoare în stivă este reținută valoarea parametrilor efectivi, în cazul transmiterii prin referință, în stivă este salvată adresa variabilelor.

Funcția *main()*

Un program C este compus dintr-o ierarhie de funcții, orice program trebuind să conțină cel puțin funcția *main()*, prima care se execută la lansarea programului C. Codul funcției se găsește între acolade.

```
void main(void){
    /* programul principal aferent fiecărui program C */
}
```

Definirea și declararea unei funcții

Pentru a putea fi utilizată într-un program, o funcție trebuie să fie definită (sau declarată) și apelată. **A defini** un subprogram, înseamnă a scrie efectiv corpul funcției. Trebuie avut grijă de alegerea locului unde se definește subprogramul (dacă el nu a fost anterior declarat atunci definiția trebuie să aibă loc înainte de apelul acestuia). **A declara** un subprogram înseamnă a-l anunța. Un subprogram nedeclarat nu poate fi folosit. Definiția unui subprogram ține loc și de declarație.

Definiția unei funcții are următoarea formă:

```
tip_rezultat_returnat  nume_funcție  (lista_parametri_formali){//antetul
                                                                    funcției
    definirea variabilelor locale
    prelucrări           // instrucțiuni
    întoarcere rezultat
}                        // între { } este implementat corpul funcției
```

Funcțiile nu pot fi definite imbricat (ca în Pascal). Definiția unei funcții trebuie să corespundă declarației funcției. Fiecare argument al funcției (matematice) capătă un nume, devine un parametru formal, chiar dacă în declarația funcției făcută înaintea (și distinct de) definiției funcției numărul acestora a fost declarat variabil (prin construcția sintactică „...”).

Dacă funcția nu returnează nici un rezultat (se mai numește funcție *void*) și nu primește parametri, definiția va fi:

```
void nume_funcție (void){//antetul funcției
    definirea variabilelor locale
    prelucrări           // instrucțiuni
}                        // între { } este implementat corpul funcției
```

O funcție *void* fără parametri poate prelucra variabilele globale și cele locale. Dacă mai multe funcții *void* fără parametri trebuie să aibă acces la aceleași date, acestea trebuie să fie variabile globale.

Exemplul următor conține definiția funcției 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;
}
```

Declarația unei funcții se face prin precizarea **prototipului funcției** (antetul acesteia): *tip_rezultat_returnat* *nume_funcție* (*lista_parametri_formali*);

Prototipul (*declarația funcției*) implicit este:

int nume_funcție(*void*);

Prototipul unei funcții trebuie să apară în afara tuturor funcțiilor și înaintea oricărui apel al său. Prin intermediul prototipului compilatorul știe cum trebuie apelată funcția. Compilatorul află detalii referitoare la tipul returnat de funcție, numărul și tipul parametriilor de intrare (efectivi), numele funcției folosite ca și parametru (în cazul apelurilor indirecte de funcții prin pointer). Informațiile sunt extrem de utile mai ales atunci când compilatorul trebuie să genereze cod pentru o funcție care a fost apelată înaintea definiției sale în codul sursă. De asemenea, se poate întâmpla ca, în cazul proiectelor de aplicații care conțin mai multe fișiere sursă și biblioteci (fișiere *header* și sursă *C*), o funcție să fie apelată într-un fișier sursă (scris de un programator) și să fie definită într-altul (scrisă de alt programator). În aceste situații, fișierul în care este făcut apelul funcției trebuie să „*includă*” fișierul header cu declarația funcției și este compilat separat de fișierul care conține definiția funcției, în final după faza de link-editare rezultând fișierul executabil ca un „tot unitar”.

Apelul unei funcții:

nume_funcție(*lista_parametri_efectivi*) /*poate apare ca operand într-o expresie, dacă funcția returnează un rezultat*/

La apelul unei funcții (subprogram), se realizează transferul controlului din (sub)programul apelant către subprogramul apelat și se execută corpul acestuia, după care se revine în funcția apelantă, la instrucțiunea următoare apelului. O funcție poate fi apelată, dacă în fața apelului există definiția sau cel puțin declarația funcției. Pentru a realiza acest lucru, compilatorul

păstrează pe stiva de date aferentă funcției apelate adresa de revenire în (sub)programul apelant (vezi subcapitolul 10.2.2). În cazul funcțiilor *void* fără parametri, apelul se face prin:

`nume_funcție();` //instrucțiune care efectuează o acțiune

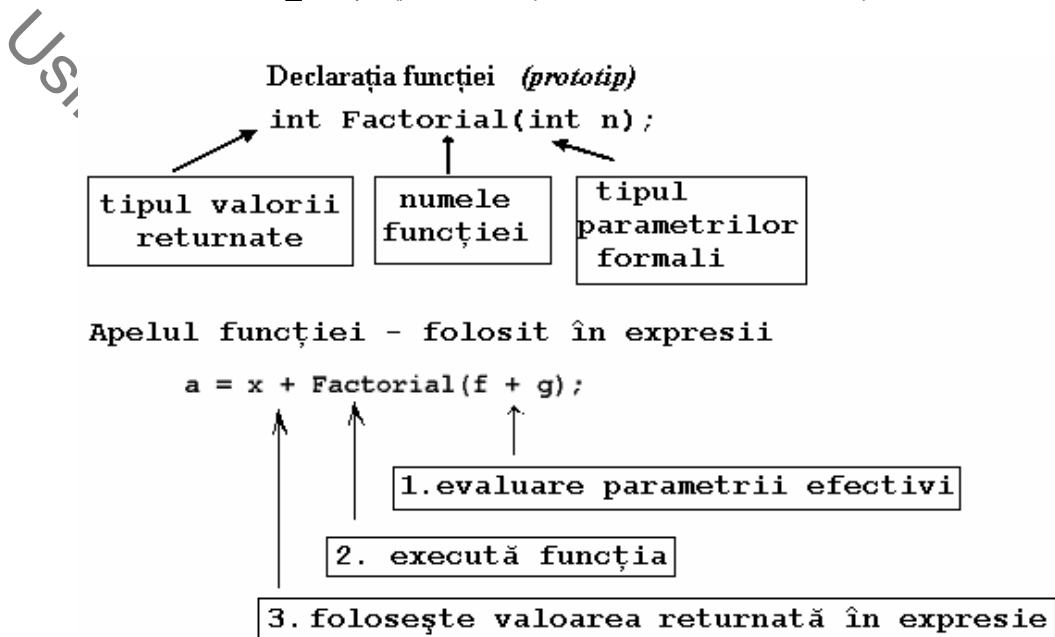


Figura 10.6. Exemplu de declarație și apel a funcției *Factorial*

În continuare este prezentat un exemplu de program C care efectuează operații aritmetico-logice asupra unor variabile locale și globale și afișează pe ecran rezultatul. Acest program C este translatat în cod asamblare LC-3, arătându-se și tabela de simboluri generată de compilator [Patt03].

Codul sursă C

```
#include <stdio.h>
int inGlobal;

main()
{
    int inLocal;    /* variabile locale funcției main */
    int outLocalA;
    int outLocalB;

    /* inițializare */
    inLocal = 5;
    inGlobal = 3;
```



```
/* efectuarea operațiilor */
outLocalA = inLocal++ & ~inGlobal;
outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

/* afișare 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)

; funcția *main* și inițializarea 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 știe, stiva de date asociată unei funcții scrise în limbajul C reprezintă o zonă de memorie unde sunt stocate toate variabilele locale și parametrii aferenți respectivei funcții ("activation record"). Fiecare funcție C are propriul context păstrat în această structură de date specială. Stiva de date se asociază în mod dinamic fiecărei funcții în curs. Așadar, o funcție poate avea la un moment dat mai multe stive (instanțe), doar una fiind însă activă. Spre exemplu, recursivitatea se poate implementa facil în C tocmai datorită acestei caracteristici.

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,z;
    /* Corpul funcției */
    .
    .
    return y;
}

```

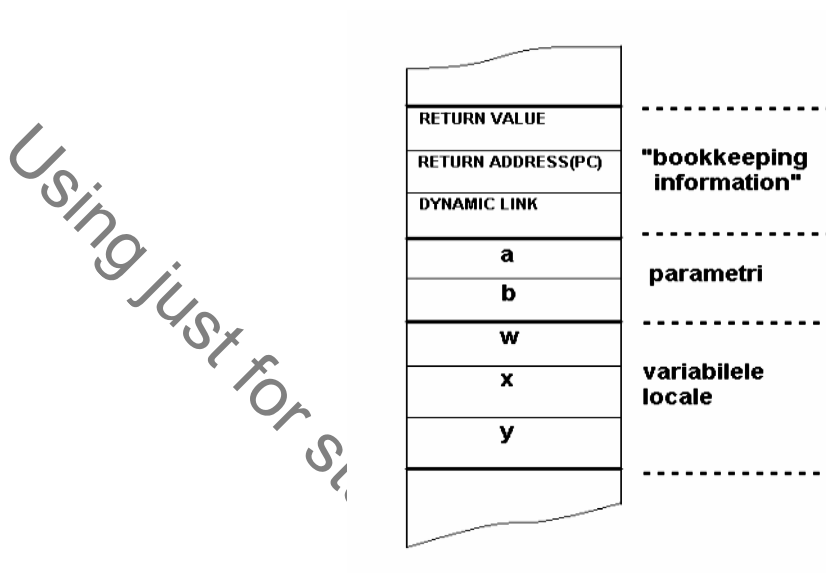


Figura. 10.7. Structura stivei de date asociate funcției NoName

RETURN VALUE: aici se plasează valoarea variabilei *y*, chiar înainte de revenirea din funcție. Acest câmp există și 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. În continuare, se va considera că registrul R6 va conține adresa de început a stivei de date asociată funcției respective.

Așadar funcția *NoName* procesează asupra parametrilor trimiși de către funcția apelantă (*a*,*b*) respectiv asupra variabilelor locale ale acesteia (*w*,*x*,*y*). Firește, ea trimite rezultatele (*y*) către funcția apelantă prin stiva de date curentă. În continuare se consideră că atunci când funcția *NoName* este apelată, pointerul la stivele de date (R6, aici) va pointa la începutul stivei de date aferente funcției.

Pentru a înțelege implementarea apelurilor/revenirilor funcțiilor, 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 funcției */
    return y;
}

```

Stiva de date începe la o locație de memorie determinată de către proiectanții sistemului de operare și crește înspre adrese crescătoare. 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 registrul R6 pointează la începutul ei. În traducerea apelului unei funcții, compilatorul generează automat cod mașină pentru a înscrie o stivă de date în memorie. În traducerea revenirii dintr-o funcție apelată în funcția apelantă, compilatorul generează automat cod pentru preluarea stivei de date din memorie.

Apelul și revenirea se fac în 4 pași, așa cum se prezintă în continuare :

1) Apelul funcției NoName

Se face prin asignarea `b=NoName(a,10)`; Stiva de date a funcției `main()` respectiv a funcției `NoName(a,10)` sunt prezentate în figura următoare.

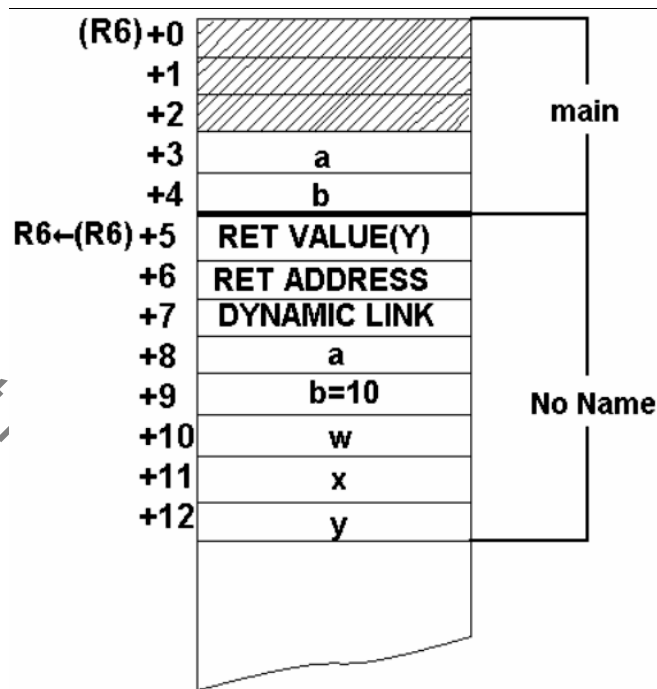


Figura 10.8. Stivele de date asociate funcțiilor main() și NoName(a,10)

Secvența compilată a apelului funcției 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 funcție, R7←PCnext și PC←(NoName)

PCnext: ld R8, (R6)5
st R8, (R6)4

2) Startarea funcției apelate (NoName)

Începe cu instrucțiunea care salvează în stiva de date a funcției NoName adresa de revenire în funcția principală main(). Adresa de revenire

se află stocată în registrul R7 (conține adresa următoare a instrucțiunii JSR NoName)

st R7, (R6)1

3) Sfârșitul funcției 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;

$R7 \leftarrow \text{RET ADDRESS}$

ld R6, (R6)2;

$R6 \leftarrow$ adresa de început a stivei de date aferente funcției main ().

RET;

$PC \leftarrow$ adresa de revenire în funcția main().

4) Revenirea în funcția apelantă (main)

JSR NoName; pe această instrucțiune se face revenirea (la finele execuției acesteia)

PCnext: ld R8, (R6)5;

$R8 \leftarrow$ valoarea lui y din funcția NoName

st R8, (R6)4;

se face asignarea : $b = \text{NoName}(a, 10)$.

Obs.: Stivele de date asociate funcțiilor C sunt structuri de tip tablou, având o adresă de bază stocată în registrul R6 și un număr (variabil) de elemente. Având în vedere frecvența deosebită a accesării acestor structuri de date, modul de adresare indexat ($R_{\text{bază}} + \text{offset}$) este esențial în facilitarea manipulării datelor prin aceste structuri de tip tablou [Vin06].

10.3. EXERCITII ȘI PROBLEME

1. Completați cu instrucțiunile C corespunzătoare în tabelele din partea dreaptă astfel încât din punct de vedere logic să se realizeze același lucru ca și în stânga (cel descris de secvența asamblare LC-3).

a)

<pre> LDR R0, R6, #3 LDR R1, R6, #4 STR R0, R6, #4 STR R1, R6, #3 </pre>	<pre> main() { int v1; int v2; int temp; } </pre>
--	---

b)

<pre> 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 </pre>	<pre> main() { int v3; int v4; int v5; } </pre>
---	---

c)

<pre> AND R0, R0, #0 ADD R0, R0, #15 STR R0, R6, #4 FL1 AND R0, R0, #0 STR R0, R6, #3 FL1_2 LDR R0, R6, #3 </pre>	<pre> main() { int i; int j; int k; } </pre>
---	--

	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:

- Ce reprezintă declarația unei funcții? Care este scopul declarării?
- Ce este prototipul unei funcții?
- Ce reprezintă definiția unei funcții?
- 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 secțiuni de cod de mai jos (una în C, cealaltă în asamblare LC-2) trebuie să realizeze același lucru. Completați cele trei spații goale cu câte o instrucțiune 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 citește de la tastatură două caractere și afișează maximumul dintre ele prin intermediul funcției 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) Realizați tabela de simboluri generată de compilator pentru programul dat.
- b) Presupunând că de la tastatură se introduc caracterele 'A' și 'Q' și că valoarea inițială a registrului R6 (*stack pointer*) este 0x4000, descrieți stiva de date aferentă funcțiilor *main()* și *Max()* în câteva momente importante: (inițial când doar funcția *main* este activă, după citirea celor două caractere de la tastatură când și funcția *Max* devine activă – la apel, când funcția *Max* este activă înainte de revenire când există rezultatul disponibil, după predarea rezultatului de către *Max* când doar funcția *main* a mai rămas activă, înainte de terminarea programului).
- c) Realizați translatarea în cod asamblare LC-3 a funcției *Max* din C.

11. INTRODUCERE ÎN *RECURSIVITATE*. COMPARAȚIA DINTRE *RECURSIV* ȘI *ITERATIV* ÎN ALEGEREA ALGORITMULUI DE REZOLVARE A PROBLEMELOR. AVANTAJE / DEZAVANTAJE

11.1. SCOP ȘI COMPETENȚE NECESARE

Capitolul de față își propune o introducere în mecanismul recursivității, bazându-se pe noțiunile anterioare de funcție (subprogram, subrutină, etc) și 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 și respectiv de a aplica anumiți algoritmi în prelucrarea structurilor de date. Scopul lucrării este de cunoaștere și înțelegere a mecanismului recursivității (după unii autori recursivitatea pare „magică”, dincolo de înțelegere). Se va analiza comparativ modul de rezolvare recursiv și iterativ al unor probleme. Se va lămurii folosind diferite exemple necesitatea aplicării condiției de consistență și în ce situații este utilă rezolvarea recursivă a unor probleme.

11.2. RECURSIVITATEA

Reprezintă o **noțiune fundamentală a informaticii** și face parte din domeniul general al algoritmilor și structurilor de date. Constituie o tehnică de programare care permite o **exprimare extrem de concisă și 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 însăși noțiunea care se definește. La scrierea unui algoritm recursiv este suficient să gândim ce se întâmplă la un anumit nivel, pentru că la orice nivel se întâmplă exact același lucru. Recursivitatea în programare – a apărut după anii '80 odată cu limbajele de nivel înalt moderne Algol, Pascal, C, Fortran și 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) și 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 și 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 și crește 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 traducerea **apelului unei funcții**, compilatorul generează automat cod mașină pentru a **înscris o stivă de date în memorie**. În traducerea **revenirii 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;
}
```

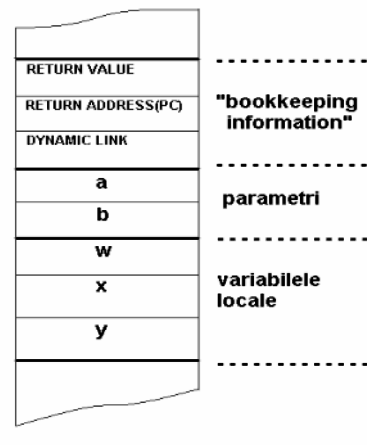


Figura 11.1. Structura stivei de date asociate funcției *NoName* în cadrul arhitecturii CC-3

RETURN VALUE: aici se plasează valoarea variabilei *w*, chiar înainte de revenirea din funcție. Acest câmp există și 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 pași. Altfel va apărea eroarea „*Stack Overflow!*”).

■ Exemplu de funcție **inconsistentă**:

$$\text{incons}(n) = \begin{cases} 1, & \text{dacă } n = 0 \\ n \cdot \text{incons}(n+1), & \text{dacă } n > 0 \end{cases}$$

întrucât nu pot fi calculate valorile funcției pentru $n > 0$.

Pentru a înțelege implementarea apelurilor / revenirilor funcțiilor, se consideră cel mai simplu și reprezentativ exemplu de subprogram recursiv factorialul unui număr natural n .

$$\text{fact}(n) = n! = n \cdot (n-1)!$$

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 (<i>fact(n)</i>)	fact(4)
Return Address	
Dynamic Link	
n (4)	fact(3)
Return Value (<i>fact(n-1)</i>)	
Return Address	
Dynamic Link	fact(2)
n-1 (3)	
Return Value (<i>fact(n-2)</i>)	
Return Address	fact(1)
Dynamic Link	
n-2 (2)	
Return Value (<i>fact(1) = 1</i>)	fact(1)
Return Address	
Dynamic Link	
n-3 (1)	

↓ Creșterea stivei

Figura 11.2. Imaginea stivei pentru calculul valorii $\text{fact}(4)$ în implementarea recursivă

În continuare sunt ilustrate două exemple în care se impune rezolvarea recursivă a problemei (algoritmul de *căutare binară* și respectiv problema *turnurilor din Hanoi*). Avantajul, în cazul primului exemplu îl reprezintă și timpul logaritm de execuție față de cel polinomial (liniar) dacă rezolvarea problemei ar fi fost iterativă (căutare directă – parcurgere tablou și 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 pași:

0. Dacă lista are cel puțin 2 studenți atunci se execută pașii, 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 sfârșitul 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* (destinație) și *B* (manevră). Pe tija *A* se găsesc *n* discuri de diametre diferite, așezate 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 așeze un disc cu diametrul mai mare peste un disc cu diametrul mai mic.

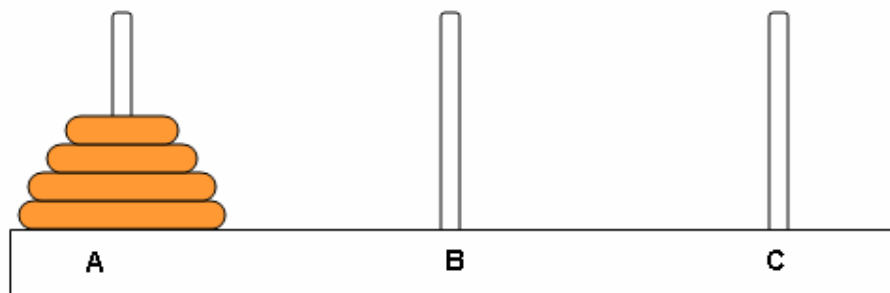
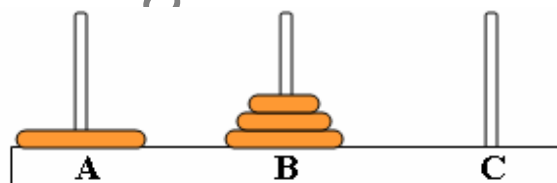
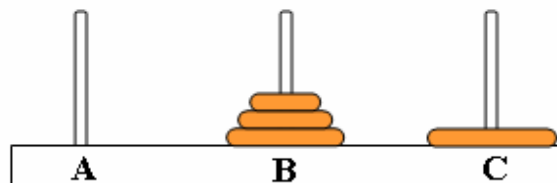


Figura 11.3. Turnurile din Hanoi – Configurația inițială

Mută primele *n-1* discuri de pe A pe B



Mută discul cu diametru cel mai mare pe C



Mută cele *n-1* discuri de pe B pe C

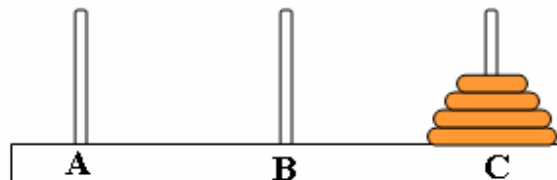


Figura 11.4. Turnurile din Hanoi – Descompunerea problemei în subprobleme

De fapt, prima subproblemă este identică cu problema inițială, cu deosebirea că sunt mai puține discuri de mutat și tijele și-au schimbat ordinea (A este sursă, B este destinație și C manevră).

- "Mută primele $n-1$ discuri de pe A pe B."

A doua subproblemă, și ea coincide în mare parte cu problema inițială, deosebirea reprezentând-o numărul mai mic de discuri de mutat și ordinea tijelor (B este sursă, C este destinație și A manevră).

- "Mută cele $n-1$ discuri de pe B pe C."

Se observă astfel caracterul recursiv al problemei. Condiția de ieșire 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);
}
```

Observatii:

- Pentru orice algoritm *recursiv* există unul *iterativ* care rezolvă aceeași 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 și memorie, timpul necesar de calcul pentru algoritmul iterativ fiind mult mai mic.
- **Consecință:** Nu întotdeauna alegerea unui algoritm recursiv reprezintă un avantaj.

- Exemplificare: Șirul lui Fibonacci – șir 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 calcului elementului $fib(n)$ din cadrul acestui șir recurent este prezentată mai jos (stânga):

```
#include<stdio.h>
int fib(int n);
main () {
    int in;
    int numar;
    printf("Care termen din șir?");
    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:**

- Șirul Fibonacci este utilizat în probleme de sortare, căutare și strategii de joc (IA).
- A fost folosit în matematică de Edmond Lucas pentru a arăta că $2^{127}-1$ este număr prim [Cor90].

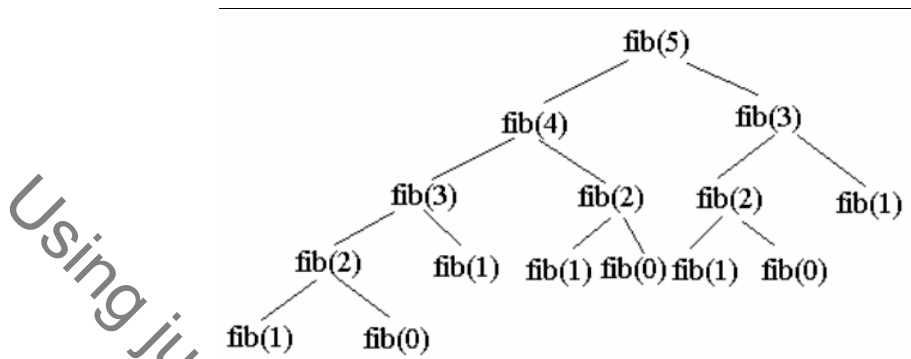


Figura 11.5. Ineficiența recursivității în cascadă – calculul unor termeni de foarte multe ori

▪ **Dezavantaje (*recursivitate în cascadă*):**

- Pentru calculul lui $fib(n)$ este necesar să se cunoască $fib(n-1)$ și $fib(n-2)$. Parametrii acestor funcții sunt depuși în stivă. Procedul continuă până este calculat $fib(n-1)$, apoi se reia calculul lui $fib(n-2)$. Acest lucru este extrem de inefficient 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 și 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 **creșterea 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 și 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 și la nivel de instrucțiune mașină – *fine grain*.

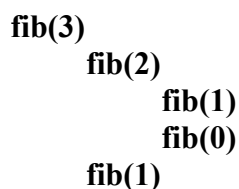


Figura 11.6. Trace-ul apelurilor funcției $fib(3)$

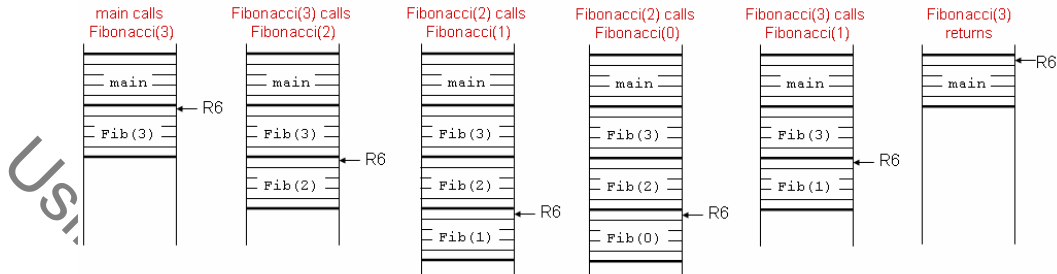


Figura 11.7. Variația stivei (și a indicatorului de stivă – R6) în cazul apelurilor funcției `fib(3)`

11.3. IMPLEMENTAREA RECURSIVITĂȚII LA NIVELUL STIVEI DE DATE

Esența implementării recursivității constă în manipularea stivelor de date asociate funcțiilor dinamice (adică funcțiilor în curs de execuție la un moment dat). La implementarea algoritmilor este uzual să asociem obiecte locale funcțiilor, precum variabile, constante, tipuri, obiecte care nu au semnificație în afara funcției respective. **Definițiile parametrilor funcțiilor recursive, a variabilelor locale și globale sunt fundamentale pentru execuția programului** [Patt03]. Astfel, variabilele folosite pentru a parcurge valorile permise componentelor unei soluții trebuie să fie locale funcției, altfel nu se generează corect soluția. Definițiile bine alese pot optimiza spațiul de memorie alocat pe stivă și timpul de execuție. De fiecare dată când o funcție este apelată recursiv, pe stiva sistemului se crează un nou set de variabile locale. Deși noile variabile locale au același nume cu cele existente înainte de activarea funcției, valorile lor sunt distincte, și 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 funcției `Fib(int n)`, cu referire la stivele de date accesate [Vin03, Patt03].

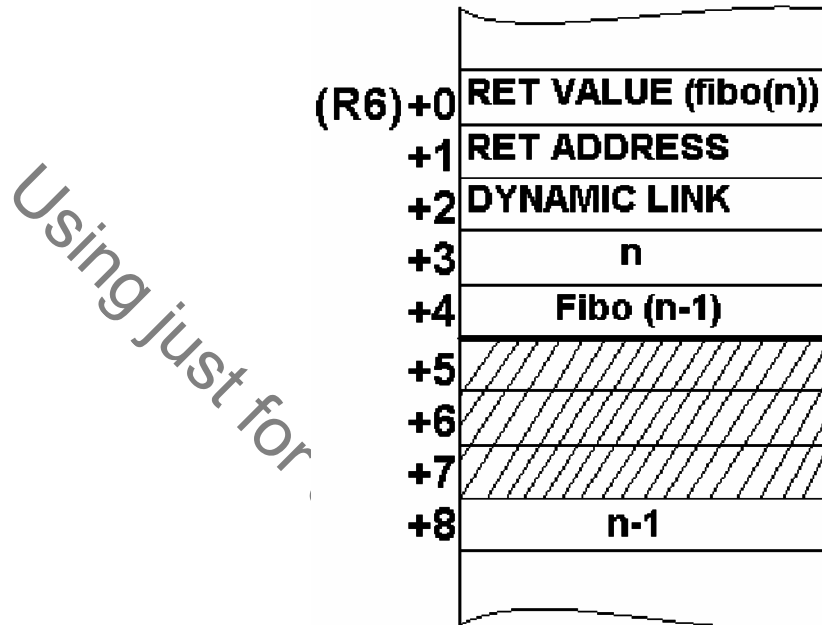


Figura 11.8. Stiva de date asociată funcției Fib(n)

Fib:

STR R7, R6, #1;	salvează PC revenire în stiva de date
LDR R0, R6, #3;	$R0 \leftarrow$ valoare "n"
BRz Fib_end;	dacă $n=0$ atunci se încheie recursivitatea, $fib(0)=1$
ADD R0, R0, # -1	
BRz Fib_end;	și dacă $n=1$ se încheie recursivitatea, $fib(1)=1$
; temp = fib(n-1)	
LDR R0, R6, #3;	se calculează n-1 $R0 \leftarrow n$
ADD R0, R0, # -1;	$R0 \leftarrow n-1$
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;	$R0 \leftarrow$ 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;      pregătește stiva de date a lui fib(n-2)
ISR 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 și stack
                      pointer)

LDR R6, R6, #2
RET

```

Obs: Legătura între arhitectura unui microprocesor și aplicațiile scrise în limbaje de nivel înalt este una complexă și extrem de subtilă. Există microarhitecturi de calcul optimizate în mod special în vederea rulării eficiente a unor clase de aplicații bine precizate scrise în anumite limbaje de nivel înalt (ex. microprocesoare Java) [Patt03].

- **Concluzie:** Este necesară o comparație între cele două moduri (*iterativ* și *recursiv*) de rezolvare a problemei date, nu doar văzut avantajul introdus de simplitatea și 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 aceleiași alternative.

Recursivitatea liniar repetitivă

Este un exemplu de recursivitate liniară. Apelul unei funcții de numește 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 și h reprezintă funcții de parametru x .

Recursivitate neliniară

În funcțiile recursiv neliniare, două sau mai multe apeluri recursive pot apare în aceeași 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 și al memoriei ocupate. Avantajul formei recursive îl constituie în primul rând facilitatea și eleganța

scrierii programelor (de exemplu, comparați programul care rezolvă *problema turnurilor din Hanoi* în cele două variante – iterativ și recursiv). În alegerea căii recursive sau nerecursive (iterative) de rezolvare a unei probleme, programatorul trebuie să stabilească prioritățile în realizarea programului, analizând complexitatea problemei, naturalețea exprimării, ușurința proiectării și testării programului, eficiența în execuție. Astfel, dacă problema e de complexitate redusă, însă se cere eficiență maximă, se va alege varianta nerecursivă. Varianta recursivă este preferată acolo unde înlocuirea complexității presupune tehnici de programare speciale, algoritmul pierzându-și naturalețea.

Algoritmul general de eliminare a recursivității liniare este următorul:

- i. Se declară o stivă, care se inițializează ca fiind vidă. Pe această stivă urmează să se salveze parametrii formali, și variabilele locale funcției recursive.
- ii. Cât timp condiția de continuare a recursivității e îndeplinită, se efectuează următoarele:
 - iii. Se salvează pe stivă valorile actuale pentru argumentele funcției recursive și 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 și se calculează valoarea dorită (după apelul funcției recursive) – eventual se execută instrucțiunile de după apelul funcției recursive, apoi se trece la pasul ii.

11.5. EXERCITII ȘI PROBLEME

1. La fiecare apel recursiv al unui subprogram, în memoria stivă sunt salvate:
 - a) adresa de revenire, valorile variabilelor locale și a parametrilor transmiși prin referință.
 - b) adresa de revenire și valorile variabilelor globale.

- c) adresa de revenire, valorile variabilelor locale și a parametrilor transmiși prin valoare și adresele parametrilor transmiși prin referință.
- d) adresa de revenire, valorile variabilelor locale și 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 și numai dacă nu conține mai mult de un autoapel.
- c) Orice subprogram recursiv se poate implementa și 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 și 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 și parametru:
- $$S = 1 - 2^2 + 3^2 - 4^2 + \dots \pm n^2$$
6. a) Adresa de revenire dintr-o funcție recursivă este întotdeauna aceeași ? 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("Introduceți 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:

- Ce afișează programul dacă se citesc de la tastatură perechile:
 - 4 9
 - 27 5
 - 1 3
- Ce calculează funcția Power() ?
- Descrieți un instantaneu (o imagine a zonei de activare a stivei de date aferentă funcției Power) în cazul apelului Power(11,7).

	Zona de Activare pentru Power
11	
7	
	← R6
	Zona de Activare pentru Power

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:

- Transformați funcția recursivă într-una iterativă care să realizeze același lucru. Se presupune că sigma este apelată doar cu parametrii pozitivi.
- Considerând exact 1 KB de memorie contiguă disponibilă exclusiv pentru stiva de date aferentă apelului recursiv al funcției Sigma, și știind că adresele și 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 ȘI 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** și **tablourile**. Pentru început se reamintește (vezi capitolul 10) că o variabilă reprezintă o locație de memorie caracterizată prin adresă și conținut. **Pointerul¹⁸ 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:

- **Pointerii către** (adrese de) **date** – conțin adresa unei variabile.
- **Pointerii către** (adrese de) **funcții** – conțin adresa codului executabil a unei funcții. Deși în C, o funcție nu este o variabilă este posibil a se referi pointeri la funcții. Aceștia pot fi atribuiți, plasați în tablouri, transmiși ca parametrii altor funcții.
- **Pointerii 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 cunoaște exact tipul de date (adresa și conținutul) care va fi instanțiat în mod dinamic la execuție.

Înțelegerea și utilizarea corectă a pointerilor este esențială pentru acuratețea și 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 și poate fi folosit cu sensul de referință, indicator de adresă, localizator.

- **Pointerii permit** (facilitează) rezervarea (alocarea) **dinamică a memoriei** (doar în momentul execuției se cunoaște adresa reală și conținutul unei variabile și 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 execuției aplicației – vezi cazul tablourilor de dimensiuni foarte mari. Alocarea de zone de memorie și eliberarea lor în timpul execuției programelor permite gestionarea optimă a memoriei de către programe. Cu ajutorul pointerilor pot fi create diferite modalități de organizare a datelor în structuri, care cresc sau descresc pe durata execuției programelor (un exemplu ar fi stiva de date aferentă funcțiilor) [Patt03].
- **Pointerii pot îmbunătăți eficiența anumitor rutine.**

Pe lângă aceste facilități, pointerii impun însă și responsabilitate atunci când sunt folosiți. Pointerii neinițializați dar utilizați în operații (sau care conțin valori neadevate) pot determina blocarea sistemului de operare. De asemenea, pointerii folosiți în mod incorect implică erori greu de depistat.

Întrucât pointerul reprezintă adresa unei zone de memorie rezultă că orice operație cu pointeri presupune citirea / scrierea în memorie (iar în cazul celor neinițializați – într-o zonă de memorie necunoscută) [Zah04]. Efectul operației de citire nu este atât de grav din punct de vedere al funcționalității sistemului ci doar strict din punct de vedere al aplicației, care va folosi astfel o valoare eronată. Problema cea mai neplăcută la folosirea pointerilor neinițializați poate apare la atribuirea (asignarea) unei valori conținutului 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 execuției programului. Exemplul următor ilustrează un caz de pointer neinițializat.

```
void main(void)
{
    int x, *p;
    x = 12;
    *p = x;
    ...
}
```

Această secvență de cod ilustrează o eroare destul de frecventă și 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, deși 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 și mai complexă (folosirea tablourilor de pointeri sau liste înlănțuite), efectul acestei greșeli poate deveni „neplăcut” și 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 și 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, $\text{int } *p, \text{int } *q; p=q$). 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 și 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 ȘI 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 aceeași prioritate ca toți ceilalți operatori unari din C.

În declarația $\text{int } *p$; tipul int stabilește faptul că p conține adrese de zone de memorie alocate datelor de tip int . Declarația lui p poate fi

interpretată astfel: $*p$ reprezintă conținutul zonei de memorie spre care indică p , iar acest conținut 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 (inițializarea pointerului) se va folosi operatorul unar `&`. De exemplu, declarațiile `int a; int *p = &a;` exprimă de fapt trei operații: o declarație a variabilei a (alocare statică a memoriei – `int a;`) o declarație a unui pointer spre tipul `int` (`int *p;`) și o inițializare a lui p cu adresa variabilei a (`p = &a;`).

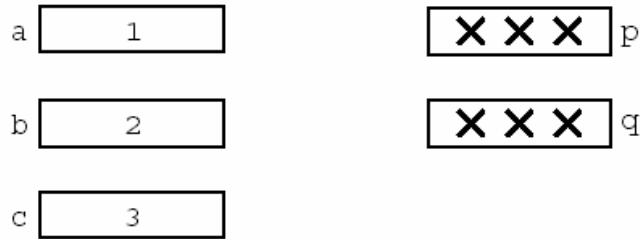
În continuare sunt prezentate câteva echivalențe de construcții de limbaj folosind pointeri.

Se consideră declarațiile:	
<code>int x, y;</code> <code>int *p;</code>	
<code>y = x + 100;</code>	<code>p = &x;</code> <code>y = *p + 100;</code>
<code>x = y;</code>	<code>p = &x;</code> <code>*p = y;</code>
<code>x++;</code>	<code>p = &x;</code> <code>(*p)++;</code>

Exemplul următor ilustrează (și grafic) instrucțiunile prin care un pointer este inițializat și cum se modifică referințele la variabile pe parcursul unei secvențe de program.

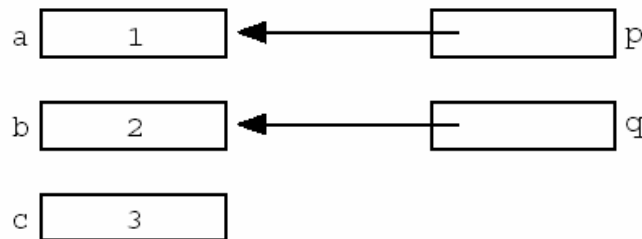
```
void Exemplu_cu_Pointer(){
    int a = 1;
    int b = 2;
    int c = 3;
    int* p;
    int* q;
```

// Conținutul memoriei în acest punct (T1) este următorul (pointerii p și q nu au fost încă inițializați).



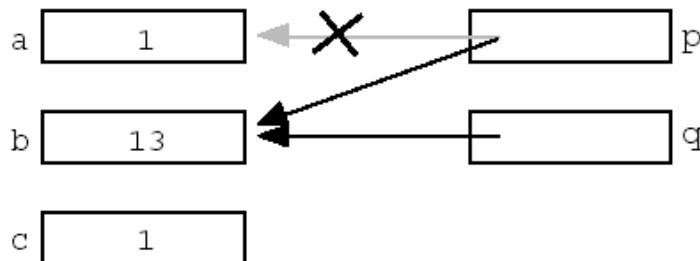
```
p = &a; // este inițializat p să indice spre variabila a, adresa lui a
        fiind cunoscută în momentul compilării
q = &b; // este inițializat q să indice spre variabila b, adresa lui b
        fiind cunoscută și ea în momentul compilării
```

// Conținutul memoriei în acest punct (T2) este următorul:



```
c = *p;
p = q; // se pierde referința anterioară a lui p (care era spre a) și va
        partaja împreună cu q aceeași referință spre b.
*p = 13; // prin modificarea efectuată asupra valorii de la locația spre
        care pointează p se modifică și *q.
```

// Conținutul memoriei în acest punct (T3) este următorul:



```
}
```

Figura 12.1. Aplicație C folosind pointeri

12.3. ALOCAREA ȘI ACCESAREA DE VARIABLE. LEGĂTURA DINTRE NIVELUL HIGH ȘI LOW VĂZUTĂ PRIN INTERMEDIUL MODURILOR DE ADRESARE

În cadrul LC-3 ISA [Patt03] **accesul la conținutul unei locații de memorie corespunzătoare unei variabile alocate static** poate fi făcut prin intermediul instrucțiunii $LD\ R_{dest}, Adresă$ (**mod de adresare direct**) – dacă se află într-un spațiu de $(+255 / -256)$ locații față de adresa instrucțiunii curente și respectiv, folosind succesiunea de instrucțiuni (1) $LEA\ R_{bază}, Adresă$ și (2) $LDR\ R_{dest}, R_{bază}, \#0$ – cu mod de adresare indexat, mai ales atunci când locația de memorie se află într-un spațiu îndepărtat cu cel puțin 256 de locații față de PC-ul instrucțiunii curente. **Accesul la conținutul unei locații spre care indică o variabilă de tip *pointer*** (alocare dinamică a memoriei) se face folosind o instrucțiune de acces la memorie în **mod de adresare indirect** $LDI\ R_{dest}, Adresă$ sau, după cum poate fi observat și în exemplul următor, printr-o succesiune de trei instrucțiuni: (1) $LEA\ R_{bază}, Adresă$, (2) $LDR\ R_{cont}, R_{bază}, \#0$ și (3) $LDR\ R_{dest}, R_{cont}, \#0$.

Aplicație 1: Accesul la conținutul unei locații spre care indică o variabilă de tip *pointer*.

Se consideră următoarea configurație arhitecturală:

Adresă memorie	Conținut locație
x3050	x70A2
x70A2	x70A3
x70A3	xFFFF
x70A4	x123B

Registrul PC indică valoarea x3010 iar instrucțiunile de la adresele x3010 la x3012 sunt ilustrate mai jos. Să se determine valoarea stocată în R6 după execuția celor 3 instrucțiuni. Pot fi înlocuite cele trei instrucțiuni cu una singură care să rezolve aceeași problemă ? Care este aceasta ?

Adresă memorie	Codificare instrucțiune
x3010	LEA R3, x3050
x3011	LDR R4, R3, #0
x3012	LDR R6, R4, #0

Instrucțiune	Conținut registrii după execuția instrucțiunii		
	R3	R4	R6
LEA R3, x3050	x3050		
LDR R4, R3, #0	x3050	x70A2	
LDR R6, R4, #0	x3050	x70A2	x70A3

Practic secvența de 3 instrucțiuni realizează: **$R6 \leftarrow \text{Mem}[\text{Mem}[x3050]]$** care poate fi codificată mai simplu printr-o singură instrucțiune: **LDI R6, x3050**, execuția însă presupunând același număr de pași (3 – determinare adresă, determinare conținut locație de la adresa respectivă care va reprezenta adresa finală de la care se va aduce operandul sau valoarea dorită).

Aplicație 2.1: Corespondența limbaj de nivel înalt –C și limbaj asamblare LC-3 folosind pointeri [Patt03].

Cele două secțiuni de cod de mai jos (una în C, cealaltă în asamblare LC-3) trebuie să realizeze același lucru. Se consideră variabilele x, p și z declarate global. Instrucțiunile scrise cu litere aldine și subliniate au fost adăugate pentru a crea corespondența corectă C – LC-3.

int x;	.ORIG x3000
int *p;	
int z;	
main() { x = 0;	LD R0, X AND R0, R0, #0 ST R0, X
.	.
x++;	LD R0, X ADD R0, R0, #1 <u>ST R0, X</u>
<u>p = &x;</u>	LEA R0, X ST R0, P
z = (*p) + 1;	<u>LDI R0, P</u> ADD R0, R0, #1 ST R0, Z HALT
}	X .BLKW 1

	P	.BLKW 1
	Z	.BLKW 1
		.END

Aplicație 2.2: Corespondența limbaj de nivel înalt –C și limbaj asamblare LC-3 folosind pointeri. Se consideră de această dată că cele două variabile sunt declarate local funcției `main()`.

```
int i;
```

```
int *ptr;
```

stocheaza valoarea 4 in memorie la locatia asociata variabilei *i*

```
i = 4;
```

retine adresa variabilei *i* in locatia de memorie asociata variabilei *ptr*

```
ptr = &i;
```

```
*ptr = *ptr + 1;
```

citeste continutul locatiei de memorie spre care indica *ptr*

stocheaza o noua valoare la adresa indicata de *ptr*

; *i* este prima variabilă declarată local (offset-ul 3 față de începutul stivei de date aferentă funcției `main()`); *ptr* este a doua variabilă (offset 4).

```
; i = 4;
```

```
AND R0, R0, #0 ; inițializare R0
```

```
ADD R0, R0, #4 ; stochează valoarea 4 în R0
```

```
STR R0, R6, #3 ; scrie conținutul lui R0 în stiva de date aferentă  
funcției main() la locația corespunzătoare  
variabilei i
```

```
; ptr = &i;
```

```
ADD R0, R6, #3 ; R0 = R6 + 3 (R0 reține 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 functiei 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;
}

```

Funcția `Swap()` este apelată din programul principal `main()` cu argumentele `valueA` egal cu 13 și `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 funcției `Swap()` rămân nemodificate în urma compilării și execuției codului.

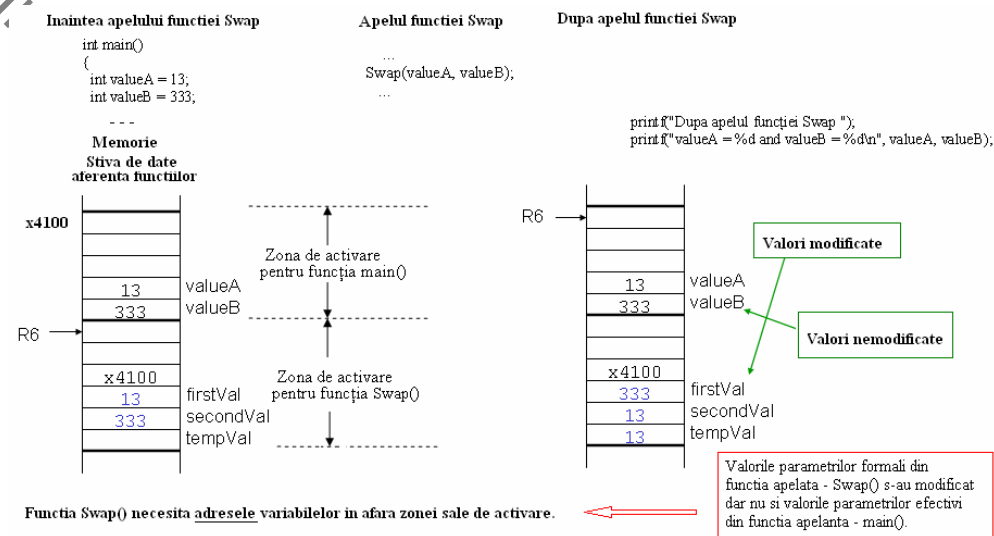


Figura 12.2. Execuția funcției `Swap` – imaginea zonei de activare înainte și după apel

Examinând stiva de date aferentă funcțiilor `main()` și `Swap()` pe durata execuției programului (vezi figura 12.2) se observă cum interschimbarea operează doar asupra copiilor locale ale parametrilor `firstVal` și `secondVal`, iar la revenirea din `Swap()` în `main()` valorile modificate sunt pierdute odată cu descărcarea de pe stivă a zonei de activare aferentă funcției `Swap()`. „Vinovat” de acest fapt este limbajul C care transmite argumentele de la nivelul funcției apelante la cel al funcției apelate prin **valoare**. „C”-ul evaluează fiecare argument care apare în apelul de funcție ca o expresie și plasează **valoarea** acesteia în locația corespunzătoare din zona de activare aferentă funcției apelate. Pentru ca funcția `Swap()` să poată modifica argumentele actuale – parametri efectivi pe care funcția apelantă `main()` îi transmite, ea (`Swap`) trebuie să aibă acces la zona de activare a funcției apelante (trebuie să acceseze locațiile la care sunt stocate argumentele pentru a modifica valorile acestora). Rezultă, în mod evident că, funcția `Swap` are nevoie de adresele variabilelor `valueA` și `valueB` din `main()` pentru a modifica valorile reținute de acestea. După cum poate fi observat din

secvența de cod de mai jos dar și din figura 12.3, acest lucru poate fi realizat folosind pointerii și 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;
}
```

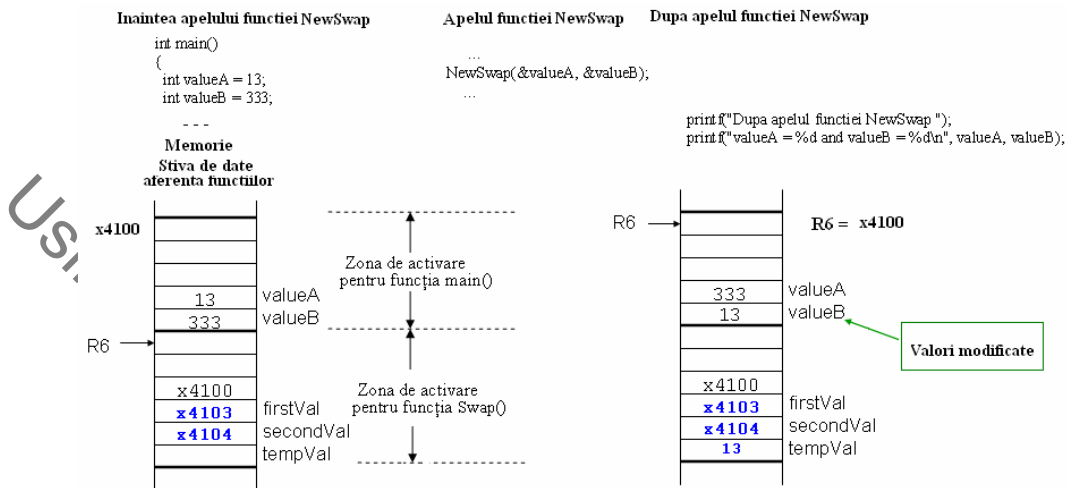


Figura 12.3. Execuția funcției *NewSwap* – imaginea zonei de activare înainte și după apel

În continuare, sunt ilustrate schimbările intervenite la nivelul funcției *Swap()*, de la începutul acestui paragraf (12.4) și din figura 12.2, pentru a realiza obiectivul inițial, cel de interschimbare a argumentelor primite. Noua variantă (revizuită) a funcției *Swap()* se va numi *NewSwap()*. Prima modificare constă în faptul că parametrii formali ai funcției *NewSwap* nu mai sunt de tip întreg (*int*) ci pointeri la întregi (*int **). În noua funcție, cei doi parametri sunt adresele de memorie a celor două variabile care se vor inversate (interschimbate). Operatorul de indirectare folosit în prototipul funcției *NewSwap* semnifică faptul că variabilele spre care se poartă vor fi interschimbate. Întrucât aceste valori se regăsesc în zona de activare a funcției apelante (*main*), la apelul funcției *NewSwap* se va folosi operatorul adresă (vezi figura 12.3).

Echivalența C – LC-3 la nivelul funcției apelante *main()* și la nivelul funcției apelate *NewSwap()* [Patt03].

Funcția *main()* urmărește interschimbarea valorilor variabilelor locale *valueA* și *valueB*, motiv pentru care transmite funcției *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 ($R0 \leftarrow x4103$). 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*** ($R1 \leftarrow 16$).

STR R1, R6, #5; se salvează în variabila *tempVal* conținutul lui R1 ($tempVal \leftarrow 13$).

; *firstVal = *secondVal;

LDR R1, R6, #4; în R1 se încarcă valoarea parametrului formal, care reprezintă **adresa variabilei *valueB*** din funcția apelantă ($R1 \leftarrow x4104$). 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*** ($R2 \leftarrow 333$).

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 ($R2 \leftarrow 13$).

STR R2, R1, #0 ; Se modifică a doua variabilă cu valoarea finală ($valueB=13$).

Diferența esențială între cele două tipuri de apeluri, prin valoare respectiv prin referință, este următoarea: **funcția apelată prin valoare nu poate modifica parametrii efectivi (actuali) din funcția care a făcut apelul neavând acces la ei, în schimb, în cazul apelului prin referință, funcția apelată, dispunând de adresa parametrilor efectivi, îi poate modifica pe aceștia** [Patt03, Neg97].

12.5. POINTERI SPRE FUNCȚII

O caracteristică generatoare de confuzii dar performantă a limbajului C este *pointerul către o funcție*. Deși o funcție nu este o variabilă, ea are o localizare în memorie care poate fi atribuită unui pointer. Adresa unei funcții este punctul de intrare în funcție. Astfel, un pointer către funcție poate fi utilizat pentru a apela respectiva funcție.

În continuare se reamintește pe scurt modul în care o funcție este compilată și apelată (vezi capitolul 10). Codul sursă este transformat în cod obiect (tabela de simboluri, analiză și sinteză) și se stabilește un punct de intrare – vezi directiva `.ORIG` (la LC-3) [Patt03] care stabilește valoarea cu care se încarcă PC-ul primei instrucțiuni din funcție. În timpul rulării programului, atunci când este apelată o funcție, acest punct de inserare este apelat de limbajul mașină (inserat pe stiva de date aferentă programului apelant – vezi figura 11.8. *Stiva de date asociată funcției Fib(n)*). Dacă un pointer conține adresa punctului de intrare, poate fi folosit pentru a apela acea funcție. Adresa unei funcții se obține utilizând numele unei funcții fără nici o paranteză sau argumente.

Pentru a declara o variabilă pointer către o funcție, numele pointerului trebuie precedat de simbolul `*`; de exemplu, **`int (*pf)(long)`** este un pointer către o funcție care are un parametru de tip `long` și întoarce un întreg. Practic, se declară o variabilă de tip pointer, care este capabilă să rețină un pointer către o funcție ce returnează un întreg. Considerând o funcție 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 conține următoarele linii de cod:

```
int rez, n;
```

```
cin>>n;
```

```
pf = cif_control;
```