Limbajul MIPS - Lectia 1 - Generalitati:

Calculatorul este format din unitatea centrala, memorie si periferice; ele comunica intre ele prin magistrale (de adrese, date, semnale).

Calculatorul pe care il avem in vedere are urmatoarele caracteristici:

Unitatea centrala (procesorul) are un numar de registri; fiecare registru este desemnat in programul sursa (MIPS) prin $numar sau $nume, iar intern (in programul scris in limbaj masina) printr-un numar intreg (anume numarul care apare in scrierea $numar).

Prezentam cativa registri si rolul lor:

Nume	Numar	Rol
$zero	$0	are mereu valoarea 0
$at	$1	rezervat pentru asamblor
$v0, $v1	$2,$3	val. prod. de o expr.sau ret. de o fct.
$a0 - $a3	$4 - $7	parametri actuali
$t0 - $t7,$t8,$t9	$8 - $15,$24,$25	val.temporare (nerestaurate de apeluri)
$s0 - $s7	$16 - $23	val.temporare (restaurate de apeluri)

Subliniem ca toti reg. de mai sus in afara de $zero pot fi modificati de utilizator, dar se recomanda ca ei sa fie folositi doar in scopurile mentionate - astfel asiguram compatibilitatea cu programe scrise de altii si evitam efecte imprevizibile. De ex. $at este folosit de compilator in instructiunile prin care se expandeaza pseudoinstructiunile; astfel, manevrarea explicita a lui de catre utilizator poate produce efecte imprevizibile.
In programele obisnuite vom folosi cu precadere $t0 - $t9.

Memoria are 2^32 octeti, avand adrese numere succesive pe 32 biti: 0x00000000, 0x00000001, ..., 0xffffffff.

Ea se poate accesa la nivel de byte (1 octet), word (4 octeti), etc.
Datele word se pot aloca/citi/scrie doar la adrese multiplu de 4 (tentativele de a face altfel genereaza erori).

Programele se scriu in limbajul de asmablare MIPS, iar la compilare sunt traduse in limbaj masina. Programul MIPS contine instructiuni si pseudoinstructiuni. La compilare o instr. MIPS este translatata intr-o instr. masina, in timp ce o pseudoinstr. MIPS este translatata in 1 - 3 instr. masina. Instructiunile masina ocupa fiecare cate 1 word.

Memoria este impartita in zone (segmente) cu destinatii specifice - stocarea datelor statice, stiva, stocarea programului (in cod masina) executat, etc.

PCSpim este o masina virtuala.

Interfata ei contine urmatoarele panouri:
- Registers:
  afisaza continutul curent al registrilor;
  se reactuaizeaza de fiecare data cand programul se opreste din executie (de exemplu intre pasi, la rularea pas cu pas);
- Text segment:
  afisaza zona de memorie ce contine instructiunile (in cod masina ale) programului si ale sistemului (kernel - un nucleu de sistem de operare, care se incarca automat);
  fiecare instructiune ocupa 1 word si se afisaza pe cate o linie, de ex.:

```
   [0x00400000]   0x8fa40000   lw $4, 0($29)      ; 175: lw $a0 0($sp) # argc
    (a)            (b)          (c)                (d)

   (a) - este adresa din memorie a instructiunii;
         cum o instructiune are 1 word, adresele vor fi din 4 in 4;
   (b) - este codificarea hexa interna (1 word) a instructiunii;
   (c) - este descrierea mnenomica (in limbajul MIPS) a instructiunii;
   (d) - ce e dupa ";" este linia sursa (MIPS) care a generat instructiunea;
         linia este precedata de numarul ei din fisierul sursa si ":";

  o linie sursa poate contine o instructiune sau o pseudoinstructiune MIPS;
  o instr. MIPS se translateaza la compilare intr-o singura instr. masina;
    ei ii va corespunde o linie in fereastra "Text segment", care in (c) va
    contine un text MIPS identic sau echivalent cu linia din fisier;
  o pseudoinstr. MIPS se translateaza la compilare prin una sau mai multe
    instr. MIPS, care se translateaza fiecare in cate o instr. masina; in
    acest caz liniei sursa respective ii vor corespunde mai multe linii in
    fereastra "Text segment"; aceste linii vor avea in (c) mnemonicul MIPS al
    instr. generate (deci alt text ca linia din fisierul sursa); de asemenea,
    doar prima dintre linii va avea ceva dupa ";" (anume linia din fisierul
    sursa), urmatoarele nu vor avea nimic;
- Data segment:
   afisaza zona de momorie ce contine datele statice ale programului, stiva
    (programul si sistemul folosesc aceeasi stiva) si datele sistemului;
- Messages:
   afisaza mesajele PCSpim (de ex. mesajele de eroare);
- Console:
   este continuta intr-o fereastra separata si simuleaza ecranul (text al)
    masinii virtuale; interactiune programului rulat cu utilizatorul (intrari
    si iesiri) se fac in aceast afereastra.

Programele sunt continute in fisiere text cu extensia".s".
Ele sunt editate separat (de ex. cu "notepad").
Se incarca cu: File -> Open
Programele se pot rula:
 -  continuu: instructiunile se executa una dupa alta automat;
 -  pas cu pas: dupa executarea fiecarei instructiuni se asteapta o comanda
     (de obicei tasta) de continuare; intre timp se poate urmari/modifica (in
     panouri si casete de dialog) continutul memoriei si al registrilor;
Diverse comenzi care se pot da:
 F5 - lansarea executiei continue; se cere adresa instructiunii de la care sa
       inceapa rularea; valoarea care apare implicit este ok pentru
       majoritatea cazurilor asa ca se poate da ENTER;
 Ctr-C - inreruperea rularii continue si comutarea in rularea pas cu pas
      (utila de ex. daca se intra intr-un ciclu infinit);
 F10 - se executa un pas - o instructiune (instructiunea curenta);
 F11 - se executa mai multi pasi - un grup de instructiuni incepand de la cea
        curenta (se cere numarul de instructiuni);
 Ctrl+B - adaugare/eliminare breackpoint-uri (se cer adresele instructiunilor
        unde se pun breackpoint-uri); cand executia un mod continuu ajunge la
        un breackpoint, ea se intrerupe si comuta in modul pas cu pas;
 Simulator -> Set Value - modificarea continutului memoriei sau registrilor
        in timpul executiei pas cu pas (se cere numarul registrului sau
        adresa de memorie si noua valoare);

-----

Structura unui program simplu (fisier ".s"):


.data
 # declaratii date
.text
 # cod
```

```
main: # eticheta marcand punctul de start
 # cod
li $v0,10
syscall
 # o linie goala
```

Comentariile se scriu in stil C++, dar incep cu #.
Etichetele sunt nume carora la compilare li se asociaza adrese de memorie;
 plasarea intr-un loc al programului a unei etichete urmate de ":" face ca
 la compilare sa se asocieze etichetei adresa de memorie la care s-a ajuns
 cu generarea programului executabil in acel moment.
Secventa "li $v0,10", "syscall" incarca valoarea imediata 10 in registrul $v0
 si apeleaza sistemul cu syscall (care preia valoarea din $v0 ca parametru);
 efectul este terminarea executiei programului; mecanismul este analog
 apelarii intreruperii 21h cu functia 4ch in assembler sub MS-DOS.

Datele se declara astfel:

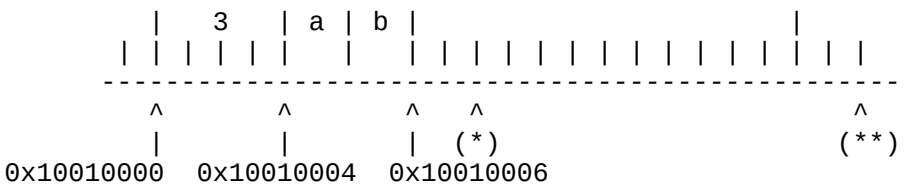
```
 nume:  tip  valori
```

unde
 "nume" este o eticheta, careia i se asociaza la compilare adresa pana la
   care s-a ajuns cu umplerea zonei de date a memoriei in acel moment;
 "tip" este un tip de date si poate fi ".byte", ".word", ".space", etc.;
 "valori" este o lista de valori compatibile cu "tip"; ele vor fi stocate in
   zona de date incepand de la adresa "nume", fiecare data ocupand o locatie
   corespunzatoare lui "tip"; valorile se pot scrie numeric zecimal (ex. 2,
   10, -1), numeric hexa (ex. 0x21 care inseamna 33 zecimal), caracter (ex.
   'A', 'b'), etc.

Exemplu:

```
  var1: .word 3
  x: .byte 'a', 'b'
  y: .space 12
```

va face ca in zona de date sa se aloce urmatoarele:



etichetei "var1" i se va asocia adresa 0x10010000 (fiind declarata cu tipul
 word, este un multiplu de 4); aici se stocheaza valoarea 3 in format intern
 de intreg fara semn pe 4 octeti; conventia de ordonare a octetilor este cea
 a masinii gazda - in cazul procesoarelor intel este little-endian, i.e.
 octetii mai putin semnificativi se afla la adrese mai mici; astfel la
 adresele 0x10010000, 0x10010001, 0x10010002, 0x10010003 vom gasi octeti
 continand respectiv valorile 3, 0, 0, 0;
etichetei "x" i se va asocia adresa 0x10010004 (aici s-a ajuns cu umplerea
 zonei de date); intrucat este declarata cu tipul byte, nu se fac alinieri
 suplimentare; incepand de aici se stocheaza caracterele 'a', 'b', i.e.
 codurile lor ASCII pe cate un octet (fiind date de cate un octet nu se pune
 problema conventiei de ordonare little-endian); astfel ,la adresa
 0x10010004 vom gasi un octet continand valoarea 0x61, iar la adresa
 0x10010005 un octet continand valoarea 0x62;
etichetei "y" i se va asocia adresa 0x10010006 (aici s-a ajuns cu umplerea
 zonei de date); nici tipul space nu impune alinieri suplimentare; valoarea

```
 de initializare 12 face ca de la adresa "y" sa se lase un spatiu de 12
 octeti;

notam urmatoarele:
daca dupa "y" am declara o eticheta word, ei i s-ar asocia adresa 0x10010012
 (pe care am notat-o cu (*)) si nu 0x10010010, cea de la sfarsitul locatiei
 space 12, deoarece tipul word impune alinierea la o ardesa multiplu de 4;
daca dupa "y" am declara o eticheta byte, ei i s-ar asocia adresa 0x10010010;
la adresa "y" (adica 0x10010006) nu putem citi/scrie un word (nu e o adresa
 multiplu de 4), dar putem citi/scrie un byte;
la adresa "y+2" (adica 0x10010008) putem citi/scrie atat un word cat si un
 byte (este o adresa multiplu de 4);

verificare practica:
scriem cu notepad programul "lab1_1.s", continand:

.data
  var1: .word 3
  x: .byte 'a', 'b'
  y: .space 12
.text
main:
li $v0,10
syscall

il incarcam cu File -> Open;
urmarim panoul "Data segment" si constatam ca de la adresa 0x10010000 s-au
 alocat word-urile 0x00000003, 0x00006261, etc.; 0x00000003 este valoarea
 de initializare indicata in program pentr "var1"; 0x00006261 este
 interpretarea ca word (tinand cont de ordinea little-endian) a celor 4
 octeti care urmeaza, adica 0x61 ('a'), 0x62 ('b'), 0, 0 (primii doi octeti
 din space 12); deci zona de date este afisata in panou ca sir de word-uri;
cu Simulator -> Reinitialize stergem memoria si registrii; astfel programul
 dispare din memorie si putem incarca altul;

Subliniem: etichetele declarate dupa ".data" joaca rolul variabilelor din
 limbajele de programare obisnuite, dar aici numele respective nu desemneaza
 locatii ci adrese de inceput (asemeni numelor de vectori din C); astfel,
 "var1+1" inseamna adresa 0x10010000+1=0x10010001, nu valoarea 3+1=4.

-----

Instructiunile se codifica intern pe cate 1 word, intr-unul dintre formatele
 R, I, J - detalii in lectia 2.

Stilul de lucru: datele se incarca din memorie in regisri, se opereaza asupra
 lor lucrand doar cu registrii, apoi rezultatele se scriu in memorie;
 singurele instructiuni care acceseaza memoria sint cele de transfer intre
 memorie si registri; instructiunile care fac calcule opereaza doar cu
 registri; prezentam cateva instructiuni si formatul lor intern:

- instructiuni de transfer memorie <-> registri:
  codificarea interna se face in formatul I:
                                          op = codul operatiei;
b: 31  26 25 21 20 16 15             0   rs, rt = codurile a doi registri
  |  op  | rs  |  rt |    val         |  val = o valoare imediata (i.e. care
  ------------------------------------    in prog. sursa este scrisa ca
    6 b    5 b   5 b       16 b           atare iar la compilare este
                                          inglobata in codul instructiunii,
                                          nu stocata ca o data obisnuita)
Sintaxa:

 lw  rt, adr  # in reg. "rt" se incarca un word incepand de la adresa "adr"
 lb  rt, adr  # in octetul low al reg. "rt" se incarca un byte citit din mem.
```

```
              #  incepand de la adr. "adr"; semnul se extinde la restul reg.
 sw  rt, adr  # in mem.se scrie incepand de la adr."adr" word-ul din reg."rt"
 sb  rt, adr  # in mem.se scrie incepand de la adr."adr" byte-ul low din reg.
              #  "rt"
```

in aceasta scriere:
numele "lw", "lb", "sw", "sb" defineste valoarea campului "op" din formatul
  I, anume: 0x23 (lw), 0x20 (lb), 0x2b (sw), 0x28 (sb);
"rt" este un registru care se scrie sub forma $nume sau $cod (ex.$t0 sau $8),
  iar codul sau va fi stocat in campul "rt" din formatul I;
"adr" este o expresie a carei evaluare produce o adresa de memorie; poate fi:
 imm - valoare imediata (desemnand adresa respectiva); se stocheaza ca atare
       in campul "val" din formatul I;
 eticheta - adresa desemnata este adresa asociata la compilare etichetei;
       aceasta adresa se stocheaza in campul "val" din formatul I;
 eticheta +/- imm - adresa desemnata este adresa asociata la compilare
       etichetei +/- valoarea imediata; aceasta adresa se stocheaza in
       campul "val" din formatul I;
 (rs) - un registru (scris sub forma $nume sau $cod) intre paranteze;
       adresa desemnata este continutul registrului (determinat deci la
       momentul executiei); codul registrului se stocheaza in campul "rs"
       din formatul I;
 imm(rs) - adresa desemnata este valoarea imediata (care poate fi si nula sau
       negativa) + continutul registrului (determinat deci la momentul
       executiei); in formatul I campul "rs" va contine codul registrului
       iar campul "val" valoarea imediata;
 eticheta +/- imm(rs) - adresa desemnata este adresa asociata la compilare
       etichetei +/- valoarea imediata + continutul registrului;
desi nu scrie in cartea [1], merge si:
 eticheta(rs) - adresa desemnata este adresa asociata la compilare etichetei
       + continutul registrului;

Notam ca doar forma "lw/lb/sw/sb rt, imm(rs)" este compilata intr-o
 singura instructiune masina, restul fiind translatate la fel ca si
 pseudoinstructiunile.

Utile sunt si pseudoinstructiunile urmatoare:

```
 li  reg,  val      # incarca in reg. "reg" val. imediata "val" (privita ca
                    #  word)

 move rdest, rsrc   # copiaza valoarea din reg. "rsrc" in reg "rdest"

 la  reg,  var      # incarca in reg. "reg" adresa variabilei (de fapt val.
                    #  etichetei) cu numele "var"
```

Exemplu practic:

Scriem programul "lab1_2.s" (care permuta continutul a doua variabile):

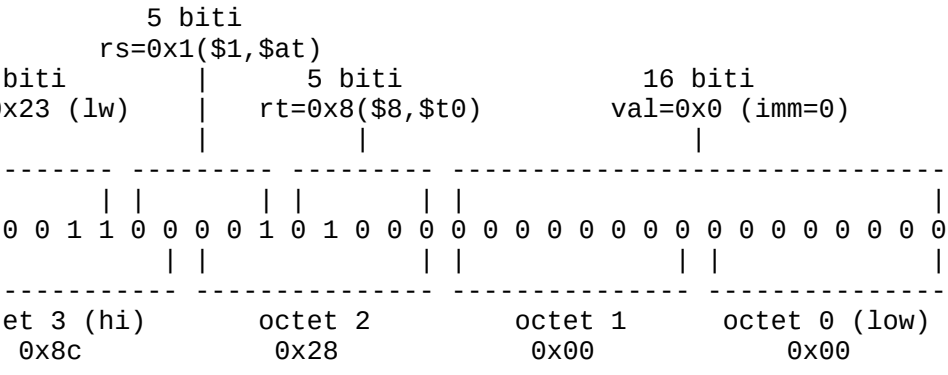
```
#swap
.data
 x:.word 1
 y:.word 2
.text
main:
 lw $t0,x
 lw $t1,y
 sw $t0,y
 sw $t1,x
li $v0,10
syscall
```

Il incarcam (File -> Open) si constatam (privind panoul "Text segment") ca

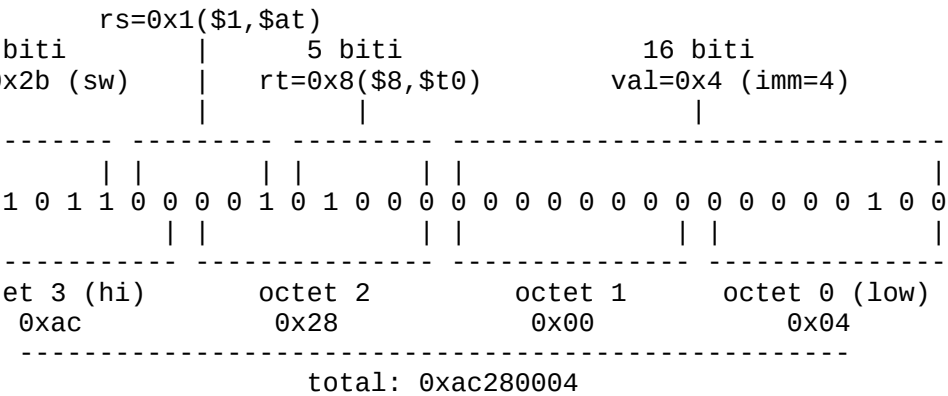
pentru secventa "lw $t0,x", "lw $t1,y", "sw $t0,y", "sw $t1,x" s-a generat:

```
[0x00400024]    0x3c011001  lui $1, 4097     ; 7: lw $t0,x
[0x00400028]    0x8c280000  lw $8, 0($1)
[0x0040002c]    0x3c011001  lui $1, 4097     ; 8: lw $t1,y
[0x00400030]    0x8c290004  lw $9, 4($1)
[0x00400034]    0x3c011001  lui $1, 4097     ; 9: sw $t0,y
[0x00400038]    0xac280004  sw $8, 4($1)
[0x0040003c]    0x3c011001  lui $1, 4097     ; 10: sw $t1,x
[0x00400040]    0xac290000  sw $9, 0($1)
```

De exemplu pentru "lw $t0,x" s-a generat secventa "lui $1, 4097",
"lw $8, 0($1)"; cu "lui $1, 4097" se incarca numarul 4097, adica 0x1001 in
jumatatea hi a lui $1 (registrul $at rezervat asamblorului) completand
jumatatea inferioara cu 0; astfel reg. $1 va contine 0x10010000 care este
(se constata privind panoul "Data segment") adresa asociata etichetei "x";
apoi "lw $8, 0($1)" incarca in reg. $8 (adica $t0) un word de la adresa
continuta in $1, adica de la adr. lui "x";
Codificarea interna (pe 1 word) a instr. "lw $8, 0($1)" este 0x8c280000;
intr-adevar, avem (pe biti):



Analog, codificarea interna a instr. "sw $8, 4($1)" este 0xac280004:



Rulati pas cu pas (cu F10) programul, urmarind in panoul "Data segment" cum
se modifica variabilele "x", "y" (de fapt locatiile word care incep de la
etichetele respective).

Varianta a programului in care datele se initializeaza la exeutie:

```
.data
 x:.space 4
 y:.space 4
.text
main:
```

```
 li $t0,1
 sw $t0,x
 li $t0,2
 sw $t0,y
# de aici e ca inainte
 lw $t0,x
 lw $t1,y
 sw $t0,y
 sw $t1,x
li $v0,10
syscall
```

Varianta a programului in care folosim adresarea indexata:

```
.data
 x:.word 1
 y:.word 2
.text
main:
 la $t2,x      # incarca in $t2 adresa corespunzatoare lui "x"
 lw $t0,0($t2) # incarca in $t0 word-ul aflat la distanta 0 de x, adica 1
 lw $t1,4($t2) # incarca in $t1 word-ul aflat la distanta 4 de x,
               # adica la distanta 0 de y, adica 2
 sw $t0,4($t2)
 sw $t1,0($t2)
li $v0,10
syscall
```

Urmatorul proram ilustreaza alte particularitati legate de stocarea datelor:

```
.data
 x:.word 2,65536    # adica 0x02, 0x10000
 y:.space 12
.text
main:
 la $t0,y      # $t0 primeste adr. lui y, care este adr. lui x + 8
 li $t1,5      # $t1 primeste valoarea 5
 sw $t1,($t0)   # I
 sw $t1,4($t0)  # II
 sw $t1 -4($t0) # III
 sb $t1,-3($t0) # IV
li $v0,10
syscall
```

ilustram modul cum se modifca memoria (am subliniat cu "=" zona modificata):

```
 x               y
 |   2   | 65536 |   0   |   0   |   0   | initial
 |2|0|0|0|0|0|1|0|0|0|0|0|0|0|0|0|0|0|0|0|
 ----------------------------------------
 x               y
 |   2   | 65536 |   5   |   0   |   0   | dupa I
 |2|0|0|0|0|0|1|0|5|0|0|0|0|0|0|0|0|0|0|0|
 ----------------=========----------------
 x               y
 |   2   | 65536 |   5   |   5   |   0   | dupa II
 |2|0|0|0|0|0|1|0|5|0|0|0|5|0|0|0|0|0|0|0|
 ------------------------=========--------
 x               y
 |   2   |   5   |   5   |   5   |   0   | dupa III (obs. ca scrierea unui
 |2|0|0|0|5|0|0|0|5|0|0|0|5|0|0|0|0|0|0|0| word a alterat 4 octeti de la adr.
 --------=========------------------------ respectiva, astfel si 1 a dev. 0)
 x               y
 |   2   | 1285  |   5   |   5   |   0   | dupa IV (obs. ca scrierea unui
```

```
 |2|0|0|0|5|5|0|0|5|0|0|0|5|0|0|0|0|0|0|0|  byte a alterat doar 1 octet de la
 ----------===----------------------------  adr. respectiva, astfel nu s-a
                                            modificat 5-ul de la adr y)
```

notam ca n-ar fi mers "sw $t1,-3($t0)", caci adr. destinatie nu e aliniata la multiplu de 4;

instr. "sb $t1,-3($t0)" a fost codificata intern 0xa109fffd; intr-adevar:

```
              5 biti
           rs=0x8($8,$t0)
   6 biti         |      5 biti               16 biti
 op=0x28 (sb)     |   rt=0x9($9,$t1)        val=0xfffd (imm=-3)
    |             |          |                    |
 ----------- --------- --------- --------------------------------
 |         | |       | |       | |                              |
 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
 |             | |             | |             | |              |
 --------------- -------------- -------------- ---------------
  octet 3 (hi)       octet 2         octet 1      octet 0 (low)
      0xa1            0x09            0xff            0xfd
      ------------------------------------------------------
                        total: 0xa109fffd
```

codul -3 s-a obtinut dupa regula "(not 3) + 1" trecand astfel succesiv prin codurile (de 16 biti): 0000000000000011 (3) -> 1111111111111100 (not 3) -> 1111111111111101 ((not 3) + 1)

- instructiuni de ramificare si salt;

instructiuni de ramificare :

```
 b  eticheta  # ramificare neconditionata la eticheta respectiva;

 beq/blt/ble/bgt/bge/bne  rs,  rt,  eticheta
   # testeaza continuturile reg. rs si rt (scrisi sub forma $nume sau $cod)
   #  verificand daca avem respectiv rs=rt,rs<rt,rs<=rt,rs>rt,rs>=rt,rs!=rt;
   #  verificarea se face considerand continuturile ca intregi cu semn;
   #  daca da, executia ramifica la eticheta respectiva;
   #  daca nu, executia trece la instructiunea urmatoare logic;

 bleu/bltu/bgeu/bgtu  rs,  rt,  eticheta
   # ca mai sus, verifica daca avem rs<=rt,rs<=rt,rs>=rt,rs>rt;
   # continuturile reg. sunt considerate intregi fara semn;
```

toate cele de mai sus pseudoinstructiuni in afara de "beq", care se codifica in format I, cu campurile:
op = 0x04;
rs,rt = codurile reg. respectivi;
val = nr.de instr. peste care trebuie sa sara executia, calculat la momentul compilarii facand diferenta dintre adr. instr. "beq" si adr. asociata etichetei si impartind la 4; find pe 16 biti, se poate sari peste maxim 2^15-1 instructiuni;

este important sa stim daca o configuratie de biti continuta intr-un registru este considerata la comparare intreg cu sau fara semn; de exemplu config. 0xffffffff inseamna 2^32-1 ca intreg fara semn si -1 ca intreg cu semn; deci 0xffffffff > 1 ca intreg fara semn si 0xffffffff < 1 ca intreg cu semn.

pentru a compara cu 0 putem folosi reg. $zero sau niste instr. de ramificare specifice - vezi lectia 2;

instructiuni de salt:

```
 j  eticheta  # salt neconditionat la eticheta respectiva
```

are efect asemanator cu "b eticheta", dar se codifica intern altfel, anume in format J:

```
b: 31  26 25                        0
  |  op  |           val            |
  ---------------------------------
    6 b             26 b
```

op = codul operatiei, aici 0x02;
val = o valoare imediata, aici este numarul de ordine al instructiunii (word-ului din memorie) la care se sare (calculat la momentul compilarii din adresa asociata etichetei); fiind pe 26 biti,se poate sari intr-o zona de 2^26 instructiuni, deci este mai flexibil ca la "b etichetei";

```
 jr  rs  # salt neconditionat la adresa continuta in registrul rs (scris sub
         #  forma $nume sau $cod)
```

codificarea interna este:

```
b: 31  26 25 21 20            6 5    0
  |  0   | rs  |       0       |  8   |  rs = codul reg. rs
  -------------------------------------
    6 b    5 b        15 b        6 b
```

Exemple:

Programul "lab1_3.s" ilustreaza cateva fenomene legate de ramificare si salt:

```
.data
 x:.word 2
.text
main:
 li $t0,2
 lw $t1,x
 beq $t1,$t0,egal
 nop
 j sfarsit
 egal:
 nop
 sfarsit:
li $v0,10
syscall
```

Comentarii:
- incarcam si urmarim panoul "Text segment";
- "nop" este instructiunea de efect nul; se codifica intern cu toti cei 32 biti 0; am folosit-o ca o instructiune manechin depre care sa stiu sigur ca nu se expandeaza (nu e pseudo) - astfel pot calcula usor nr. de octeti peste care se sare in zona de cod;
- "beq $t1,$t0,egal" se codifica intern (format I) 0x11280003; intr-adevar:

```
              5 biti
           rs=0x9($9,$t1)
   6 biti        |      5 biti              16 biti
 op=0x4 (beq)    |   rt=0x8($8,$t0)       val=0x3 (3)
    |            |         |                  |
 ----------- --------- --------- ------------------------------
 |         | |       | |       | |                            |
 0 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
 |             | |             | |             | |             |
 --------------- --------------- --------------- ---------------
  octet 3 (hi)       octet 2         octet 1      octet 0 (low)
```

```
      0x11            0x28            0x00            0x03
      ------------------------------------------------------
                        total: 0x11280003
```

obs. ca val=3, deci s-a calculat ca trebuie sa se sara 3 instructiuni (word-uri), adica peste "beq $t1,$t0,egal", "nop" si "j sfarsit"; daca in loc de "nop" si "j sfarsit" foloseam doua pseudoinstructiuni, nr. de instructiuni peste care trebuia sa se sara (i.e. nr. de instructiuni generate) era altul si mai greu de anticipat la scrierea programului.

- "j sfarsit" se codifica intern (format J) 0x08100010; intr-adevar:

```
   6 biti                          26 biti
 op=0x2 (j)                     val=0x100010
    |                                |
 ---------- ------------------------------------------------------
 |         | |                                                    |
 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
 |             | |             | |             | |             |
 --------------- -------------- -------------- ---------------
  octet 3 (hi)       octet 2         octet 1      octet 0 (low)
      0x08            0x10            0x00            0x10
      ------------------------------------------------------
                        total: 0x08100010
```

privind adresele instructiunilor (in panoul "Text segment"), constatam ca instructiunea la care se sare, "li $v0,10" (care se transcrie la compilare in "ori $2, $0, 10"), incepe de la adresa 0x00400040 (asociata etichetei "sfarsit"), fiind astfel al 0x400040 / 0x4 = 0x100010 - lea word din zona de memorie, adica valoarea lui "val";

- rulam pas cu pas (cu F10) si constatam ca dupa " beq $t1,$t0,egal" se executa ultimul "nop";
- in loc de "j sfarsit" putem folosi o secventa de tip:

```
      la $t2, sfarsit
       ...
      jr $t2
```

iar atunci in $t2 se va pune adresa, nu numarul de ordine al word-ului;

Programul "lab1_4.s" calculeaza maximul a doua numere:

```
#z:=max(x,y)
.data
 x:.word 2
 y:.word 1
 z:.space 4
.text
main:
 lw $t0,x
 lw $t1,y
 blt $t0,$t1,et1
 sw $t0,z
 j sfarsit
et1:
 sw $t1,z
sfarsit:
li $v0,10
syscall
```

Programul "lab1_5.s" sorteaza trei numere x, y, z simuland metoda bubble sort cu doi registri (se compara/interschimba (x,y), (y,z), (x,y)):

```
#sort(x,y,z)
.data
```

```
 x:.word 3
 y:.word 1
 z:.word 2
.text
main:
 lw $t0,x
 lw $t1,y
 ble $t0,$t1,et1
 sw $t0,y
 sw $t1,x
et1:
 lw $t0,y
 lw $t1,z
 ble $t0,$t1,et2
 sw $t0,z
 sw $t1,y
et2:
 lw $t0,x
 lw $t1,y
 ble $t0,$t1,et3
 sw $t0,y
 sw $t1,x
et3:
li $v0,10
syscall
```

Programul "lab1_6.s" calculeaza maximul dintr-un vector de numere.

Avem nevoie de instructiuni pentru incrementarea unui indice. Putem folosi:

```
 add  rd,  rs,  rt  # rd,rs,rt sunt tergistri; calculeaza rd := rs + rt

 addi rt,  rs, imm  # rt,rs sunt registri, imm este o valoare imediata;
                    # calculeaza rt := rs + imm
```

Programul este urmatorul:

```
.data
 v:.word 2, 1, 3, 2, 3  # vectorul
 n:.word 5              # nr. de elemente ale vectorului
 i:.space 4             # indice (putem folosi si un registru)
 max:.space 4           # aici vom stoca valoarea maxima
.text
main:
#initializari
 li $t0,0
 sw $t0,i    # i:=0
 lw $t0,v
 sw $t0,max  # max:=v[0]
#ciclu
intrare:
 lw $t0,i
 addi $t0,$t0,1
 sw $t0,i            # i:=i+1
 lw $t1,n
 bge $t0,$t1,iesire  # daca i>=n iesim din ciclu
 add $t0,$t0,$t0
 add $t0,$t0,$t0     # acum $t0=4*i, adica offsetul in octeti al lui v[i]
                     #  fata de v
 lw $t0,v($t0)       # $t0:=v[i]
 lw $t1,max
 ble $t0,$t1,et
 sw $t0,max
et:
```

```
 j intrare
iesire:
li $v0,10
syscall
```

Programul de mai sus poate fi scris in mai multe variante - de ex. putem folosi pentru indice si/sau nr. de elemente un reg. dedicat si sa nu mai tot incarcam/salvam aceste valori in memorie.

Exercitii:
---

1. Program care calculeaza maximul a trei numere.
2. Program care ordoneaza trei numere x, y, z simuland metoda sortarii prin selectie cu doi registri (se compara/interschimba (x,y), (x,z), (y,z)). Sa se foloseasca cat mai putine instructiuni - de ex. cand incarcam o valoare intr-un reg. sa folosim cat mai mult acel reg.
3. Program care calculeaza suma elementelor unui vector de numere.
4. Program care cauta un numar x intr-un vector de numere si pune in y valoarea 1/0 dupa cum x apare/nu apare in vector.
5. Program care sorteaza un vector de numere.

Bibliografie:
~~~~~~~~~~~~~

1."Organizarea si proiectarea calculatoarelor - interfata hardware/software", John L. Hennessy, David A. Patterson, ed. All, 2002, cap. 3 si anexa A

DANIEL DRAGULICI
septembrie, 2006
actualizat: 4 noiembrie 2006
~~~~~~~~~~~~~

Limbajul MIPS - Lectia 2 - Sintaxa, Date, Instructiuni:

A. Sintaxa limbajului, directive de asamblare si tipuri de date:

Limbajul MIPS este case sensitive (compilatorul face distinctie intre literele mari si mici).

Atomii lexicali din care se compune un program pot fi:
- identificatori: sunt succesiuni de litere, cifre, underscore ("_") si punct ("."), care nu incepe cu o cifra;
  exemple corecte: abc, a.b.c, a21.3_, .abc, _..a;
  exemplu gresit: 1abc;
  identificatorii sunt folositi in principal la numele etichetelor (care atunci cand refera date sunt analoagele variabilelor);
  unii identificatori sunt cuvinte rezervate (de exemplu nume de instructiuni) si nu pot fi folosite pentru a numi entitati definite de utilizator - de exemplu: b, j, add;
- comentarii: tin de la "#" si pana la sfarsitul liniei curente; semnul "#" poate aparea oriunde intr-o linie, iar compilatorul va ignora textul de "#" pana la sfarsitul liniei respective; evident, daca vrem sa inseram mai multe linii de comentariu, fiecare trebuie sa inceapa cu "#"; stilul comentariilor este analog limbajului C++ (doar ca incepe cu "#", nu "//");
- numere intregi zecimale sau hexa, asemanator ca in limbajul C: 268, -268, 0x10c, -0x10c;
- constante caracter: caracter intre apostroafe, ex: 'a', 'A', '1', ' '
- cosntante string: succesiuni de caractere intre ghilimele, ex: "Ab c 23".

In constantele caracter sau string putem folosi urmatoarele constructii ce desemneaza caractere speciale: \n (caracterul line feed sau newline, cod zecimal 10), \t (caracteru tab), \" (caracterul "); exemple: '\n', '\t', '\"', "A\n\n32 \t as\"xy".

Un program este o succesiune de linii. O linie poate contine:
- o directiva a asamblorului; aceasta nu determina generarea de date sau cod in memorie ci spune compilatorului cum sa proceseze liniile de sub ea; efectul directivei tine pana la sfarsitul programului sau pana la intalnirea altei directive;
- o declaratie de date; aceasta determina generarea de date in memorie (declaratia contine niste valori care sunt stocate in zona de date statice a programului sau a kernel-ului);
- o instructiune sau o pseudoinstructiune; aceasta determina generarea de cod in memorie; fiecare instructiune este tradusa intr-o instructiune masina pe 1 word (4 octeti) si stocata in zona de cod (text) a programului sau a kernel-ului; fiecare pseudoinstructiune este tradusa intr-un grup de instructiuni, care sunt stocate ca mai sus.

Fiecare linie ce genereaza date sau cod in memorie poate fi etichetata; in acest scop se pune la inceputul ei (nu neaparat la inceputul fizic ci ca prim element scris in linia respectiva) o eticheta (identificator) urmata de ":"; compilatorul va asocia etichetei ca semnificatie adresa de memorie pana unde a generat date, respectiv cod, pana la momentul respectiv; astfel ea va reprezenta adresa primei date, respectiv instructiuni, care ii urmeaza.
Astfel, etichetele care semnifica adrese in zona de date sunt analoage numelor de vectori din limbajul C (care reprezeinta adresa primului element, nu locatia vectorului) si ne permit sa accesam datele invecinate printr-un mecanism analog indexarii vectorilor sau deferentierii pointerilor din C. De asemenea, etichetele care semnifica adrese in zona de cod pot fi folosite in instructiunile de ramificare si salt pentru a transfera executia la instructiunea care le urmeaza.
Putem pune o eticheta pe o linie ce nu contine nimic altceva - atunci eticheta va desemna adresa primei entitati alocate de urmatoarea linie care nu e goala. In program pot aparea si linii complet goale, iar acestea vor fi ignorate.

In cele ce urmeaza, prin analogie cu alte limbaje de programare, ne vom referi uneori la o eticheta "x" pusa in zona de date si cu termenul de "variabila", avand insa in vedere locatia care incepe la adresa respectiva. Distinctia cand "x" va insemna adresa si cand locatie se va deduce din context. Subliniem insa ca din punct de vedere tehnic, "x" are doar semnificatia unei adrese.

* Directivele sunt:

```
.extern eticheta dimensiune

## declara ca eticheta "eticheta" este globala si eticheteaza o data de
#   dimensiune "dimensiune";
#  data respectiva va fi memorata intr-o parte a segmentului de date
#   care este adresata efectiv prin registrul $gp;

.globl eticheta

## declara eticheta "eticheta" ca fiind globala;
## etichetele globale sunt vizibile si din alte fisiere link-editate cu cel
#   curent;
## de obicei eticheta "main" (punctul de incepere a executiei) trebuie
#   declarata globala; in cazurile uzuale merge insa si fara a o declara
#   globala;
## exemplu:
#    .globl main
#    .data
#    x: .word 1
#    .text
#    main:
#    li $t0,10
#    sw $t0,x
#    li $v0,10
#    syscall
## directiva ".globl main" poate fi pusa si in alte parti, de exemplu:
#     ...
#    .globl main
#    .text
#    main:
#     ...
# sau:
#     ...
#    main:
#    li $t0,10
#    .globl main
#     ...
# de multe ori este pusa imediat inainte de eticheta "main":
#     ...
#    .globl main
#    main:
#     ...

.align n

## aliniaza urmatoarea data la adresa multiplu de 2^n; daca n=0 efectul este
#   anularea alinierii automate a datelor declarate cu .half, .word, .float,
#   .double, pana la urmatoarea directiva .data sau .kdata;
## exemplu:
#    x: .word 1
#    .align 3
#    y: .word 2
#  daca 1 este alocat la adresa 0x10010000, sa va mai sari un word inainte de
#   a-l aloca pe 2, a.i. word-ul 2 se va aloca la o adr. multiplu de 8 = 2^3:
```

```
#
#     |       |       |       |
#     |1| | | | | | |2| | | |
#     -------------------------
#     ^x              ^y

.data
.data adresa
.text
.text adresa
.kdata
.kdata adresa
.ktext
.ktext adresa

## face ca entitatile urmatoare sa fie alocate in zona de date statice ale
#   programului (cazurile ".data"), zona de date statice ale kernel-ului
#   (cazurile ".kdata"), zona de cod a programului (cazurile ".text"), resp.
#   zona de cod a kernel-ului (cazurile ".ktext");
#  daca "adresa" este prezenta, alocarea entitatilor respective va incepe de
#   la adresa indicata;
## in zona de cod a programului sau kernel-ului se pot stoca doar
#   instructiuni sau date declarate cu .word;
## exemplu:
#
#    .data
#    x: .word 1      # implicit se aloca in zona de date de la adr.0x10010000
#    y: .word 2      # se aloca in zona de date de la adresa 0x10010004
#    .text
#    lw $t1, 0($t0)
#    lw $t2, 0($t0)
#    .data 0x10010010
#    z: .word 3      # se aloca in zona de cod de la adresa 0x00040010
#    t: .word 4      # se aloca in zona de cod de la adresa 0x00040014
#
# daca am fi omis adresa in directiva ".data 0x10010010" si am fi scris doar
#  ".data", word-ul 3 s-ar fi alocat la adr. 0x00040008 (adiacent lui 2), iar
#  word-ul 4 de la adr. 0x0004000c

.set at
.set noat

## activeaza (at)/dezactiveaza (noat) protestul compilatorului (generarea
#   unei erori la compilare) daca instructiunile urmatoare utilizeaza
#   explicit registrul $at (e bine ca acest registru sa nu fie utilizat
#   explicit de utilizator, altfel pot aparea efecte nedorite); am obs. ca
#   setarea implicita este "at"

* Declaratiile de date sunt:

.byte n1, n2, ...

## se aloca numerele naturale n1, n2, ... (pot fi scrise zecimal, hexa sau
#   sub forma unui caracter ASCII avand codul respectiv) in octeti succesivi;
## exemplu:
#    x: .byte 0x41, 65, 'A', 66
#  va genera:
#
#     |       |
#     |A|A|A|B|
#     ---------
#     ^x
#  adica word-ul (tinand cont de little-endian): 0x42414141
```

```
.half n1, n2, ...

## se aloca numerele naturale n1, n2, ... (pot fi scrise zecimal sau hexa)
#   in jumatati de word (perechi de octeti) succesive;
## exemplu:
#    x: .half 0x41, 321
#  va genera:
#
#     |                   |
#     |0x41|0x00|0x41|0x01|
#     ---------------------
#     ^x
#  adica word-ul (tinand cont de little-endian): 0x01410041
#   (deoarece 0x41 = 65 iar 321 = 256 + 65)

.word n1, n2, ...

## se aloca numerele naturale n1, n2, ... (pot fi scrise zecimal sau hexa)
#   in word-uri (1 word = 4 octeti) succesivi;
## exemplu:
#    x: .word 0x41, 321
#  va genera:
#
#     |                   |                   |
#     |0x41|0x00|0x00|0x00|0x41|0x01|0x00|0x00|
#     ---------------------------------------
#     ^x
#  adica word-urile (tinand cont de little-endian): 0x00000041  0x00000141

.float n1, n2, ...
.double n1, n2, ...

## se aloca succesiv numerele n1, n2, ... codificate in virgula mobila, ca
#   single (1 word)/double (2 word); in program constantele n1, n2, ...
#   trebuie scrise flotant, de exemplu 2.0, nu 2);
## exemplu:
#    x: .word 2
#    y: .float 2.0
#    z: .float 4.75
#  va genera:
#
#     |                   |                   |
#     |0x02|0x0 |0x0 |0x0 |0x0 |0x0 |0x0 |0x04|0x0 |0x0 |0x98|0x40|
#     -------------------------------------------------------------
#     ^x                 ^y                  ^z
#  adica nr. 2 reprezentat ca nr. intreg, 2 reprezentat ca nr. flotant
#   single si 4.75 reprezentat ca nr. flotant single; pe word-uri avem
#   (tinand cont de little-endian): 0x00000002,  0x40000000,  0x40980000;
#  vedem ca tipul ".word" sau ".float" e folosit de compilator ca sa stie
#   in ce format intern (intreg sau flotant) va fi stocata o aceeasi valoare
#   matematica 2;

.ascii string

## se aloca stringul fara vreun terminator la sfarsit;
## exemplu:
#    x: .ascii "ABCDEFG"
#  va genera:
#
#     |       |       |
#     |A|B|C|D|E|F|G| |
#     -----------------
#     ^x
#  putem declara in loc:
```

```
#    x: .byte 'A', 'B', 'C', 'D', 'E', 'F', 'G'
#  sau:
#    x: .byte 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47
#  sau amestecat:
#    x: .byte 0x41, 66, 67, 0x44, 'E', 0x46, 'G'

.asciiz string

## se aloca stringul adaugand la sfarsit ca terminator caracterul nul
## exemplu:
#    x: .asciiz "ABCDEFG"
#  va genera:
#
#     |       |       |
#     |A|B|C|D|E|F|G|0|
#     -----------------
#     ^x
# adica pe word-uri (tinand cont de little-endian): 0x44434241  0x00474645
# putem declara echivalent:
#    x: .byte 'A', 'B', 'C', 'D', 'E', 'F', 'G', 0
#    x: .byte 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x0
#    x: .byte 0x41, 66, 67, 0x44, 'E', 0x46, 'G', 0

.space n

## se aloca un spatiu de n octeti;
## in PCSpim se poate folosi doar in segmentul de date;
```

In general datele .half, .word, .float, .double se pot aloca/scrie/citi in memorie doar la adrese care sunt multipli ale dimensiunii tipului respectiv (2, 4, 4, respectiv 8 octeti); alocarea aliniata poate fi anulata cu ".align 0", dar instructiunile de citire/scriere in continuare nu vor functiona decat cu adrese aliniate.

Notam ca nu e obligatoriu ca orice linie ce declara date sa aibe eticheta. De exemplu succesiunea:

```
x: .word 1, 2
   .word 3
y: .word 4
```

genereaza urmatorul continut in zona de date:

```
|           |           |           |           |
|1 |0 |0 |0 |2 |0 |0 |0 |3 |0 |0 |0 |4 |0 |0 |0 |
-----------------------------------------------
^x                                  ^y
```

deci in memorie se pun succesiv word-urile 1, 2, 3, 4, dar doar adresele unde a fost stocat 1 si 4 au asociate etichete; toate word-urile pot fi insa accesate, de exemplu asa:

```
lw $t0, x   # $t0 <- 1
lw $t0, x+4 # $t0 <- 2
lw $t0, x+8 # $t0 <- 3
lw $t0, y   # $t0 <- 4
```

Succesiunea de declaratii de mai sus este echivalenta cu:

```
x: .word 1, 2, 3
y: .word 4
```

Instructiunile si pseudoinstructiunile vor fi prezentate mai tarziu.

Exemplu: aratam ca putem stoca date in zona de cod, ca putem genera la
~~~~~~~~ momentul executiei cod nou pe care apoi sa-l executam, dar trebuie
  avut grija sa nu cream cod invalid (programul se va rula pas cu pas,
  urmarind zona de cod (text) in dreptul lui y,z si zona de date in dreptul
  lui x):

```
.data
x: .word 1
.text
aaa:  sw $t1,0($t0)
  bbb:
 ccc: j continua
main:     # de aici incepe executia
lw $t0,aaa  # codul instr. "sw $t1,0($t0)" (1 word) se incarca in $t0
sw $t0,y    #  si apoi se scrie la adr. y
lw $t0,ccc  # codul instr. "j continua" (1 word) se incarca in $t0
sw $t0,z    #  si apoi se scrie la adr. z
la $t0,x
li $t1,10
j y         # executia trece la adr. y si executa instr. "sw $t1,0($t0)" si
            #  apoi "j continua" scrise acolo mai devreme; acum la adr. x
            #  este un word egal cu 10
continua: # aici se revine
j continua1
.word 20         # date alocate in zona de cod; daca nu sarim peste ele
y: .space 4      #  ("j continua1") masina va incerca sa execute ce scrie
z: .space 4      #  aici (informatia poate avea sens ca instructiune - y,z -
continua1:       #  sau nu - cazul primului word egal cu 20)
li $v0,10
syscall
##########
```

Comentarii:
- eticheta "aaa" reprezinta adresa instr. "sw $t1,0($t0)", iar etichetele
 "bbb" si "ccc" reprezinta adresa aceleiasi instr. "j continua"; cele doua
 instr. sunt instr. autentice (nu pseudoinstr.), deci ocupa fiecare cate 1
 word;
- instr. "sw $t1,0($t0)" si "j continua" nu sunt executate la inceputul
 programului, deoarece punctul de start (eticheta "main") este la
 "li $t1,10";
- in total se va executa:

```
    lw $t0,aaa
    sw $t0,y
    lw $t0,ccc
    sw $t0,z
    la $t0,x
    li $t1,10
    j y
    sw $t1,0($t0)  # copiile acestor instr. stocate la adr. y
    j continua     #  si z
    j continua1
    li $v0,10
    syscall
```

- se pot face si alte artificii, de exemplu sa stocam instructiuni in zona de
 date (in program le scriem la ".data").

Exemplu: ilustram modul de aliniere a datelor (la rulare se va urmari
~~~~~~~~ continutul memoriei):

```
.data
x1: .byte 1, 2, 3, 4, 5   # x1 va fi implicit adr. 0x10010000
x2: .half 6, 7, 8
x3: .byte 9
x4: .word 10, 11
.align 0
```

```
y1: .byte 1, 2, 3, 4, 5
y2: .half 6, 7, 8
y3: .byte 9
y4: .word 10, 11
.text
main:
li $t0,0
sb $t0,x1+5   # ok
# sh $t0,x1+5 # genereaza eroare, deoarece adr. 0x10010005 nu e para
# sw $t0,x1+5 # genereaza eroare, deoarece adr. 0x10010005 nu e multiplu de 4
sb $t0,x2     # ok
sh $t0,x2     # ok
# sw $t0,x2   # genereaza eroare, deoarece adr. 0x10010006 nu e multiplu de 4
#####
li $t0,0
sb $t0,y2    # ok
# sh $t0,y2  # genereaza eroare, deoarece adr. 0x1001001d nu e para
# sw $t0,y2  # genereaza eroare, deoarece adr. 0x1001001d nu e multiplu de 4
sb $t0,y2+1  # ok
sh $t0,y2+1  # ok
# sw $t0,y2+1 # genereaza eroare, deoarece adr. 0x1001001e nu e multiplu de 4
#####
li $v0,10
syscall
##########
```

Comentarii:
- in zona de date se va aloca:

```
 |           |           |           |           |           |           |
 |1 |2 |3 |4 |5 |  |6 |0 |7 |0 |8 |0 |9 |  |  |  |10|0 |0 |0 |11|0 |0 |0 |
 ---------------------------------------------------------------------------
 ^x1               ^x2               ^x3         ^x4
 adr. 0x10010000

 |           |           |           |           |           |
 |1 |2 |3 |4 |5 |6 |0 |7 |0 |8 |0 |9 |10|0 |0 |0 |11|0 |0 |0 |
 -------------------------------------------------------------
 ^y1            ^y2               ^y3^y4
 adr. 0x10010018

 adica: 0x04030201  0x00060005  0x00080007  0x00000009  0x0000000a  0x0000000b
        0x04030201  0x07000605  0x09000800  0x0000000a  0x0000000b
```

- ".align 0" a influentat doar alocarea datelor de la y1, y2, y3, y4, nu si succesul sau esecul operatiilor de scriere cu sb (store byte), sh (store half), sw (store word) la adrese nealiniate.

B. Registri si memorie:
==========

* Registri de uz general:

Sunt in numar de 32; fiecare are 1 word (4 octeti), este identificat intern printr-un cod intreg iar in programul sursa prin $nume sau $numar (numarul fiind codul intern); pot fi folositi/modificati oricum de catre utilizator (exceptand $zero care contine mereu 0 - tentativa de a-l modifica ramane fara efect), dar este bine sa fie folositi conform anumitor roluri precizate pentru fiecare in parte - astfel asiguram compatibilitatea cu programe scrise de altii si evitam efecte imprevizibile. Registrii de uz general sunt:

Nume	Numar	Rol
$zero	$0	are mereu valoarea 0
$at	$1	rezervat pentru asamblor

$v0, $v1	$2,$3	val. prod. de o expr.sau ret. de o fct.
$a0 - $a3	$4 - $7	parametri actuali
$t0 - $t7,$t8,$t9	$8 - $15,$24,$25	val.temporare (nerestaurate de apeluri)
$s0 - $s7	$16 - $23	val.temporare (restaurate de apeluri)
$k0, $k1	$26, $27	rezervat pentru kernel
$gp	$28	pointer global
$sp	$29	pointeaza varful stivei
$fp	$30	pointeaza cadrul curent in stiva
$ra	$31	contine adresa de intoarcere din apelul de subprogram curent

* Registrii LO, HI: fiecare are 1 word (4 octeti) si sunt folositi de operatiile de inmultire si impartire intre intregi; ei nu pot fi accesati de utilizator direct, ci doar prin instructiunile mflo, mfhi (a se vedea mai jos).

* Registrii si flag-urile coprocesorului de virgula mobila (coprocesorul 1):

Sunt in numar de 32 si sunt folositi pentru stocarea numerelor in virgula mobila; fiecare are 1 word (4 octeti), este identificat intern printr-un cod intreg 0 - 31 iar in programul sursa prin $fnumar (numarul fiind codul intern), deci prin $f0, ..., $f31; pot fi folositi/modificati oricum de catre utilizator, dar numai in instructiunile specifice de lucru cu registrii de virgula mobila.
Numerele in virgula mobila in simpla precizie se pot stoca intr-un registru $f0, ..., $f31, iar cele in dubla precizie intr-o pereche de registri de cod par-impar $f0-$f1, $f2-$f3, ..., $f30-$31 (locatia dubla va fi identificata in programul sursa prin registrul de cod par).

* Registrul PC contine mereu adresa instructiunii care urmeaza sa se execute; el este consultat sau modificat de instructiuni ca j, jal (a se vedea mai jos).

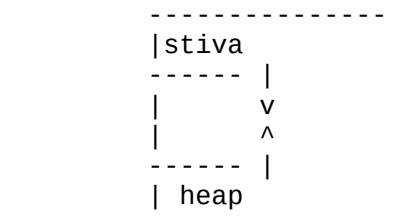
Coprocesorul de virgula mobila are de asemenea 8 flag-uri (biti) de conditie numerotati 0 - 7 (ei sunt identificati intern prin acest numar) ce sunt setati la 0/1 de instructiunile de comparare si sunt testati de instructiunile de ramificare si atribuire conditionata.

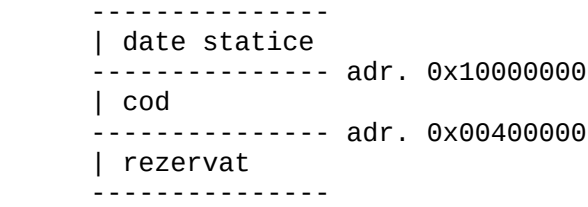
* structura memoriei folosite de program si kernel:

Memoria este impartita in zona de cod (text) a programului, zona de cod (text) a kernel-ului, zona de date statice ale programului, zona de date statice ale kernel-ului, stiva (comuna programului si kernel-ului) si heap-ul (folosit la alocari dinamice).

Zona de cod a programului/kernel-uli va contine instructiunile programului executat/kernel-ului; zona de date statice a programului/kernel-ului contine date declarate cu ".data"/".kdata" - acestea se aloca de la inceputul executiei si raman alocate pana la sfarsitul ei, similar datelor statice din limbajul C (cum sunt cele globale sau locale dar declarate cu "static"); stiva este folosita la alocarea datelor automatice (cum sunt in limbajul C cele locale fara "static") (a se vedea lectia 3) sau stocarea datelor temporare temporare (a se vedea sectiunea D); heap-ul este folosit pentru alocarea datelor dinamice (cum sunt cele alocate cu malloc in limbajul C).

Pozitionarea acestor zone de memorie este:





modul de gestiune al stivei si heap-ului este sub forma a doua stive ce cresc una spre alta; adresa octetului din varful zonei stiva este continuta in registrul $sp (el este un registru general, se poate folosi in orice alt scop, dar nu vom avea controlul zonei stiva); deci zona stiva se incarca inspre adrese mici (scade $sp) si se descarca spre adrese mari (creste $sp).

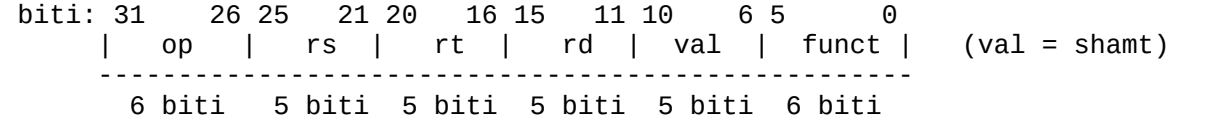
C. Instructiuni:
==========

In MIPS intalnim instructiuni si pseudoinstructiuni. O instructiune MIPS se translateaza intr-o instructiune masina, in timp ce o pseudoinstructiune MIPS se translateaza in una sau mai multe instructiuni masina.

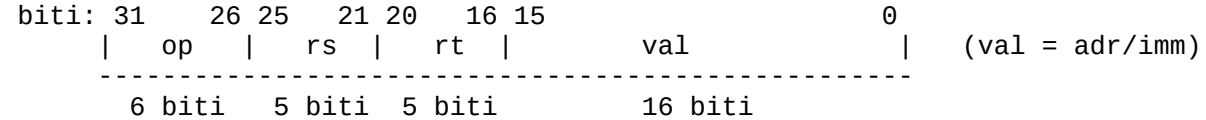
Instructiunile de calcul lucreaza doar cu registri. De aceea modul general de lucru este: se incarca datele din memorie in registri (folosind instructiunile de transfer memorie <-> registri), se fac calculele (folosind instructiunile ce lucreaza doar cu registrii), apoi eventual se salveaza rezultatele in memorie (folosind instructiunile de transfer registri <-> memorie). Singurele instructiuni ce pot accesa datele din memorie sunt cele de transfer memorie <-> registri.

Instructiunile masina ocupa fiecare cate 1 word (4 octeti) si pot avea unul din formatele I, J, R:

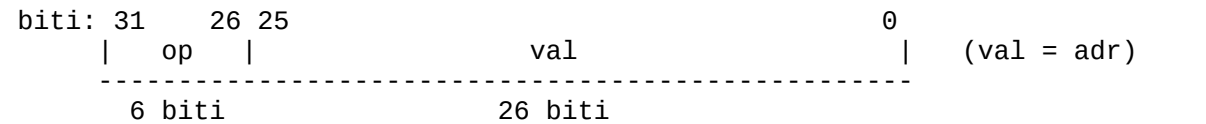
Formatul R:



Formatul I:



Formatul J:



In cele de mai sus:

op, funct = codul operatiei (cand sunt prezente ambele, ele impreuna formeaza codul operatiei);
rs, rt, rd = codurile unor registri (rs, rt sunt registri sursa, rd registru destinatie);
val = o valoare; pt. instr. in format R are semnificatia unei valori folosite la shiftari (shift amount); pt. instr. in format I are semnificatia unei adrese sau valori imediate (i.e. care in prog. sursa este scrisa ca atare iar la compilare este inglobata in codul instructiunii, nu stocata ca o data obisnuita), iar pt. instr. in format J are semnificatia unei adrese.

```
Dam in contiunare instructiunile si pseudoinstructiunile MIPS uzuale, notand:
  rs, rt, rd = registri generali, care se vor corespunde cu cei indicati in
               formatele R,I,J de mai sus;
  rsrc, rdest = registri generali (semnificand registrul sursa, respectiv
               registrul destinatie);
  reg = registru general (avand rol si de sursa si de destinatie);
  imm = valoare imediata;
la instructiunile de virgula mobila vom mai nota:
  FRdest, FRsrc, FRsrc1, FRsrc2 = registri de virgula mobila (primul avand
               rol de destinatie, ceilalti de sursa).

 In general toate instructiunile de mai jos, pana la cele din sectiunea
"Instructiuni de lucru in virgula mobila" lucreaza cu numere naturale sau
intregi si folosesc registrii de uz general. Eventualele exceptii de la
aceasta regula vor fi semnalate pentru fiecare instructiune in parte.

* Instructiuni de incarcare in registri a unor valori imediate sau adrese:

lui rt, imm

## incarca partea superioara imediat (load upper immediate);
## efectueaza: rt <- imm << 16
#    adica se incarca jumatatea de word inferioara a lui imm in
#    jumatatea de word superioara a lui rt, iar
#    jumatatea de word inferioara a lui rt se initializeaza cu 0;
#  imm trebuie sa fie din intervalul 0, ..., 2^16-1, altfel se genereaza
#   eroare la compilare (desi printre instructiunile prin care se
#   translateaza anumite pseudoinstructiuni pot aparea unele de forma:
#   lui $1, -1);
## are format I cu:  |  0xf   |   0   |   rt  |          imm           |
#                    ------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti

li rdest, imm

## incarca imediat (load immediate);
## efectueaza: rdest <- imm;
#  imm este un intreg (poate fi si negativ):
#    0, 1, -1, 0x00000001, -0x00000001, etc.
## este o pseudoinstructiune;
## de exemplu:
#    li $t0,0x00010002
#  se translateaza in:
#    lui $1, 1
#    ori $8, $1, 2
#  adica:
#    incarca 1 in cei 16 biti superiori ai lui $1;
#       astfel $1 va contine: 0x00010000
#    se pune in $8 ($t0) rezultatul disjunctiei pe biti intre $1 si 2,
#       adica intre 0x00010000 si 0x00000002, care este 0x00010002
#  in felul acesta putem incarca o valoare ce nu incape pe 16 biti (dar
#   incape pe 32 biti) folosind instructiunile ce au corespondent direct in
#   cod masina "lui" si "ori", care insa nu accepta decat valori imediate ce
#   incap pe 16 biti;
#  notam insa ca:
#    li $t0,1
#  se translateaza in:
#    ori $8, $0, 1
#  adica valoarea lui $t0 devine disjunctia pe biti intre 0x00000000 si
#    0x00000001, care este 0x00000001 - deci compilatorul cauta sa optimizeze
#    translatarea lui "li";

la rdest, eticheta
```

```
## incarca adresa (load address);
## efectueaza: rdest <- adresa asociata etichetei;
## este o pseudoinstructiune;

* Instructiuni de transfer date intre memorie si registri:

In toate instructiunile din aceasta sectiune "adr" este o expresie a carei
 evaluare produce o adresa de memorie; ea poate fi:
 imm - valoare imediata (desemnand adresa respectiva);
 eticheta - adresa desemnata este adresa asociata la compilare etichetei;
 eticheta +/- imm - adresa desemnata este adresa asociata la compilare
        etichetei +/- valoarea imediata;
 (rs) - un registru general intre paranteze; adresa desemnata este continutul
        registrului (determinat deci la momentul executiei);
 imm(rs) - adresa desemnata este valoarea imediata (care poate fi si nula sau
        negativa) + continutul registrului (determinat deci la momentul
        executiei);
 eticheta(rs) - adresa desemnata este adresa asociata la compilare etichetei
        + continutul registrului;
 eticheta +/- imm(rs) - adresa desemnata este adresa asociata la compilare
        etichetei +/- valoarea imediata + continutul registrului.

lb/lbu/lh/lhu/lw  rt,  adr

## incarca (load) octet/octet fara semn/halfword/halfword fara semn/word;
## efectueaza:
#  lb: incarca un octet de la adresa "adr" in octetul low din registrul "rt",
#        propagand bitul sau de semn b7 in restul registrului;
#  lbu:incarca un octet de la adresa "adr" in octetul low din registrul "rt",
#        propagand 0 in restul registrului;
#  lh: incarca un half de la adresa "adr" in half-ul low din registrul "rt",
#        propagand bitul sau de semn b15 in restul registrului;
#  lhu:incarca un half de la adresa "adr" in half-ul low din registrul "rt",
#        propagand 0 in restul registrului;
#  lw: incarca un word de la adresa "adr" in registrul "rt";
## incarcarea reuseste doar daca adresa "adr" este aliniata la un multiplu al
#   dimensiunii tipului (byte (1), half (2), resp. word (4));
## toate sunt pseudoinstructiuni, mai putin formele in care "adr" este
#   "imm(rs)", care au formatul I:
##                   |   op   |  rs   |   rt  |          imm           |
#                    -------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
#   unde op este respectiv: 0x20(lb)/0x24(lbu)/0x21(lh)/0x25(lhu)/0x23(lw);
## Obs: propagarea bitului de semn in cazurile lb, lh face ca config. din rt,
#   interpretata ca nr. intreg (cu semn), sa insemne acelasi lucru ca config.
#   incarcata in partea sa low;

sb/sh/sw  rt,  adr

## memoreaza (store) octet/halfword/word;
## efectueaza:
#  sb: scrie octetul low al registrului "rt" in memorie la adresa "adr";
#  sh: scrie half-ul low al registrului "rt" in memorie la adresa "adr";
#  sw: scrie word-ul din registrul "rt" in memorie la adresa "adr";
## memorarea reuseste doar daca adresa "adr" este aliniata la un multiplu al
#   dimensiunii tipului (byte (1), half (2), resp. word (4));
## toate sunt pseudoinstructiuni, mai putin formele in care "adr" este
#   "imm(rs)", care au formatul I:
##                   |   op   |  rs   |   rt  |          imm           |
#                    -------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
#   unde op este respectiv: 0x28(sb)/0x29(sh)/0x2b(sw);
```

```
ld/sd  rdest,  adr

## incarca/memoreaza cuvant dublu (64 biti, adica doua word-uri succesive);
## efectueaza:
#  ld: incarca un cuvant dublu de la adresa "adr" in perechea de registri
#      "rdest", "rdest"+1;
#  sd: scrie un cuvant dublu din perechea de registri "rdest", "rdest"+1
#      in memorie la adresa "adr";
## toate sunt pseudoinstructiuni;
## exemplu:
#   .data
#   x: .word 0x00000001, 0x00000002
#   y: .space 8
#   .text
#   main:
#   ld $t1,x
#     # se incarca word-urile 0x00000001, 0x00000002 in perechea ($t1, $t2),
#     #  adica ($9,$10);
#     # deci acum $t1=0x00000001, $t0=0x00000002
#   sd $t1,y
#     # se scriu word-urile 0x00000001, 0x00000002 din perechea ($t1,$t2) la
#     #  adr. y; deci acum la adr. y vom gasi word-urile succesive:
#     #  0x00000001  0x00000002 (adica la fel ca de la adr. x)
#   li $v0,10
#   syscall
# (gandind cuvantul dublu ca un singur numar scris in little-endian, practic
# s-a facut transferul numarului 0x0000000200000001 intre locatia de memorie
# ce incepe la adr. y si locatia-registru ($t2,$t1) ($t2 reprezentand partea
# hi si $t1 partea low);

lwl/lwr rt, adr

## incarca cuvant stanga/dreapta
## efectueaza: incarca in registrul "rt" octetii din stanga/dreapta word-ului
#              aflat la adresa "adr"; (?)
#  adresa poate fi nealiniata;
## toate sunt pseudoinstructiuni, mai putin formele in care "adr" este
#   "imm(rs)", care au formatul I:
#                    |   op   |  rs   |   rt  |          imm           |
#                    ------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
#   unde op este respectiv: 0x22(lwl)/0x26(lwr);
## exemplu:
#   .data
#   x: .word 0x01020304
#   y: .word 0x05060708
#   z: .word 0x090a0b0c
#   .text
#   main:
#   la $t0,y
#   lwl $t1,0($t0)  # $t1 va contine 0x08000000
#   lwr $t2,0($t0)  # $t2 va contine 0x05060708
#   li $v0,10
#   syscall

swl/swr rt, adr

## memoreaza cuvant stanga/dreapta;
## efectueaza: scrie octetii din stanga/dreapta registrului "rt" in memorie
                la adresa "adr"; (?)
#  adresa poate fi nealiniata;
## toate sunt pseudoinstructiuni, mai putin formele in care "adr" este
#   "imm(rs)", care au formatul I:
##                   |   op   |  rs   |   rt  |          imm           |
```

```
#                    -------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
#   unde op este respectiv: 0x2a(swl)/0x2e(swr);
## exemplu:
#   .data
#   x: .space 4
#   y: .space 4
#   z: .space 8
#   t: .space 4
#   u: .space 4
#   .text
#   main:
#   li $t1,0x01020304
#   la $t0,y
#   swl $t1,0($t0)  # la adr. y se mem. word-ul 0x00000001
#   la $t0,t
#   swr $t1,0($t0)  # la adr. t se mem. word-ul 0x01020304
#   li $v0,10
#   syscall
```

ulh/ulhu/ulw rdest, adr

```
## incarca half nealiniat/half fara semn nealiniat/word nealiniat;
## efectueaza:
#  ulh: incarca un half de la adresa "adr" in half-ul low din reg. "rdest",
#       propagand bitul sau de semn b15 in restul registrului;
#       nu este obligatoriu ca adresa "adr" sa fie aliniata  (la un multiplu
#       de 2);
#  ulhu: incarca un half de la adresa "adr" in half-ul low din reg. "rdest",
#       propagand 0 in restul registrului;
#       nu este obligatoriu ca adresa "adr" sa fie aliniata  (la un multiplu
#       de 2);
#  ulw: incarca un word de la adresa "adr" in registrul "rdest";
#       nu este obligatoriu ca adresa "adr" sa fie aliniata  (la un multiplu
#       de 4);
## toate sunt pseudoinstructiuni;
## exemplu:
#   .data
#   x: .byte 0x81
#   y: .byte 0x82, 0x83, 0x84, 0x85, 0x86
#   .text
#   main:
#   ulh $t0,y  # $t0 = 0xffff8382 (s-a propagat b15 = 1)
#   ulhu $t0,y # $t0 = 0x00008382 (s-a propagat 0)
#   ulw $t0,y  # $t0 = 0x85848382
#   li $v0,10
#   syscall
# observatii:
#  - PCSpim arata ca adresa lui y este 0x10010001, deci nu e aliniata la
#     multiplu de 2 sau 4;
#  - in scrierea ca numar a word-ului din $t0 valorile apar inversate din
#     cauza lui little-endian;
#  - in primele doua cazuri b15 este 1, deoarece in binar 8 = 1000;
```

ush/usw rsrc, adr

```
## scrie half nealiniat/word nealiniat;
## efectueaza:
#  ush: scrie half-ul low al registrului rsrc in memorie la adresa "adr";
#       nu este obligatoriu ca adresa "adr" sa fie aliniata  (la un multiplu
#       de 2);
#  usw: scrie word-ul din registrul rsrc in memorie la adresa "adr";
#       nu este obligatoriu ca adresa "adr" sa fie aliniata  (la un multiplu
#       de 4);
```

```
## toate sunt pseudoinstructiuni;

move  rdest,  rsrc

## muta;
## efectueaza: rdest <- rsrc
#   (in ciuda numelui "muta", de fapt copiaza word-ul din reg. "rsrc" in reg.
#   "rdest");
## este o pseudoinstructiune;

mflo/mfhi  rd
mtlo/mthi  rs

## muta din LO (mflo) / din HI (mfhi) / in LO (mtlo) / in HI (mthi);
## efectueaza:
#    rd <- LO (mflo) / rd <- HI (mfhi) / LO <- rs (mtlo) / HI <- rs (mthi)
#   (i.e. copiaza word-ul dintr-un registru in celalalt);
## mflo/mfhi au formatul:
#                     |   0    |       0      |   rd  |   0   |0x12/0x10|
#                     ---------------------------------------------------
#                       6 biti       10 biti   5 biti   5 biti  6 biti
#  mtlo/mthi au formatul:
#                     |   0    |   rs  |           0          |0x13/0x11|
#                     ---------------------------------------------------
#                       6 biti   5 biti         15 biti          6 biti
## LO si HI sunt registrii speciali ai unitatii de inmultire si impartire;
#  aceasta pune rezultatele inmultirii si impartirii in acesti registri
#   (a se vedea instructiunile de inmultire si impartire cu intregi de mai
#   jos); utilizatorul nu poate face alte prelucrari cu ele direct aici ci
#   trebuie sa le copieze in alti registri (de uz general); practic singurele
#   instructiuni care ii permit utilizatorului sa acceseze registrii LO si HI
#   sunt mflo, mfhi, mtlo, mthi;
## exemplu:
#   .text
#   main:
#   li $t0, 0x80000001
#   li $t1, 2
#   multu $t0, $t1  # inmultire fara semn - nu se propaga bitul de semn
#    # efectueaza: (HI,LO) <- $t0 * $t1 (fara semn)
#    # adica HI retine word-ul hi al produsului
#    #   0x80000001 * 2 = 0x0000000100000010, adica word-ul 0x00000001
#    # iar LO restine word-ul low al acestui produs, adica 0x00000010
#   mflo $t0  # acum $t0 = 0x00000010 (adica nr. 2)
#   mfhi $t1  # acum $t1 = 0x00000001 (adica nr. 1)
#   add $t2,$t0,$t1  # $t2 = 1 + 2 = 3
#    # obs. ca n-am fi putut aduna pe 1 cu 2 direct din LO si HI
#    #  ci a trebuit mai intai sa-i copiem in registri de uz general
#    #  (add lucreaza cu registri de uz general)
#   li $t0, 14
#   li $t1, 4
#   divu $t0, $t1  # impartire fara semn si fara depasire
#    # efectueaza: LO <- $t0 div $t1 (adica 3), HI <- $t0 mod $t1 (adica 2)
#   mfhi $t2       # $t2 = 2 = restul impartirii
#   bne $t2, $zero, et1 # daca $1 nu divide $t0 (rest nenul) salt la et1
#    li $t3,1
#   j et2
#   et1:
#    li $t3,2
#   et2:
#   li $v0,10
#   syscall

mfcz/mtcz  rt, reg
 (practic am observat ca merge doar pentru z = 0, 1, 2)
```

```
## muta din / in coprocesorul z;
## efectueaza:
#  mfcz: copiaza word-ul din reg. "reg" al coprocesorului z
#         in reg. "rt" al CPU;
#  mtcz: copiaza word-ul din reg. "rt" al CPU
#         in reg. "reg" al coprocesorului z;
#  coprocesorul 1 este coprocesorul de virgula mobila;
#  registrii pot fi indicati prin $cod sau $nume, avand grija se se
#   foloseasca numele specifice registrilor din UCP sau coprocesorul
#   respectiv - de ex. reg $8 al UCP este $t0 iar reg. $8 al coprocesorului
#   de virgula mobila este $f8;
## format intern:
#                    |  0x1z  |  0/4  |  rt   |  reg  |       0        |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti      11 biti
## un exemplu va fi dat pentru mfc1, mtc1 in sectiunea "Instructiuni de
#   lucru in virgula mobila";
```

Observatii:
- In toate cazurile transferul consta practic in copierea configuratiei binare (intre memorie si registri sau intre doi registri).
- In toate calculele de adrese se numara octeti, nu half-uri, word-uri; de exemplu:

```
la $t0,x
sb $t1,4($t0)
sh $t1,4($t0)
sw $t1,4($t0)
```

toate cele trei scrieri se fac la aceeasi adresa (anume la 4 octeti dupa adresa lui x) - deci plecand de la adresa lui x in toate cazurile s-au numarati 4 octeti in plus, si nu 4 half-uri (in cazul sh) sau 4 word-uri (in cazul sw);
un "adr" de forma x($t0) este asemanatoare cu indexarea vectorilor din limbajul C, x[i], numai ca in limbajul C adresa lui x[i] se calculeaza ca fiind adresa lui x deplasata cu i componente ale vectorului, nu cu i octeti.

In general instructiunile de procesare date lucreaza doar cu registri si valori imediate; de aceea, modul uzual de lucru cu memoria este: se incarca datele din memorie in registri, se opereaza cu ele in registri, se salveaza rezultatele in memorie; practic singurele instructiuni care acceaseaza memoria sunt cele descrise mai sus (si cele analoage pentru date in virgula mobila - a se vedea mai jos).

* Instructiuni de ramificare, comparatie si salt:

b eticheta

```
## ramificare neconditionata
## efectueaza: executia trece la (instructiunea aflata la adresa indicata de)
#               eticheta;
## este o pseudoinstructiune;
```

beq/bne rs, rt, eticheta

```
## ramifica la egal/diferit;
## efectueaza: daca rs = / != rt
#              atunci salt la eticheta
## are format I cu:  |0x4/0x5 |   rs  |   rt  |          depl          |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
#  unde depl este numarul de instructiuni masina (word-uri) peste care se
#   sare, numarand inclusiv instructiunea beq/bne curenta - acest numar este
#   determinat de compilator;
```

```
#  de exemplu la:
#    beq $t0, $t0, et
#    lui $t0,1
#    et:
#    lui $t0,2
#  in codificarea lui beq, depl va fi 0x0002, caci daca $t0 = $t0 se va sari
#   la "lui $t0,2", adica peste 2 instrunctiuni (numarand de la cea curenta):
#   "beq $t0, $t0, et", "lui $t0,1"

blt/bltu/ble/bleu/bgt/bgtu/bge/bgeu rsrc1, rsrc2, eticheta

## ramificare la
#   mai mic strict cu semn / mai mic strict fara semn /
#   mai mic sau egal cu semn / mai mic sau egal fara semn /
#   mai mare strict cu semn / mai mare strict fara semn /
#   mai mare sau egal cu semn / mai mare sau egal fara semn;
## efectueaza:
#   daca
#      rsrc1
#            < cu semn / < fara semn /
#            <= cu semn / <= fara semn /
#            > cu semn / > fara semn /
#            >= cu semn / >= fara semn /
#      rsrc2
#   atunci salt la eticheta
## la comparatia cu semn configuratiile din rsrc1 si rsrc2 sunt interpretate
#   ca numere naturale si comparate fara semn; la comparatia fara semn sunt
#   interpretate ca numere intregi si comparate cu semn;
#  de exemplu:
#    li $t0,0x00000000  # si ca natural si ca intreg $t0 = 0
#    li $t1,0xffffffff  # ca natural $t0=2^32-1 (>0), ca intreg $t0=-1 (<0)
#    blt  $t0,$t1,et1    # nu sare (ca intregi nu avem 0 < -1)
#    bltu $t0,$t1,et2    # sare (ca naturale avem 0 < 2^32-1)
## sunt pseudoinstructiuni;

bltz/blez/bgtz/bgez rs, eticheta

## ramificare la
#   mai mic strict / mai mic sau egal /
#   mai mare strict / mai mare sau egal
#  ca 0
## efectueaza: daca rs < / <= / > / >= 0
#              atunci salt la eticheta
## are format I cu:  |1/6/7/1 |   rs  |0/0/0/1|          depl          |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
#  unde depl este numarul de instructiuni masina (word-uri) peste care se
#   sare, numarand inclusiv instructiunea beq/bne curenta - acest numar este
#   determinat de compilator;

beqz/bnez reg, eticheta

## ramificare la egal cu/diferit de 0
## efectueaza: daca reg = / != 0 atunci salt la eticheta
## sunt pseudoinstructiuni;

bltzal/bgezal rs, eticheta

## ramificare si legatura la
#   mai mic strict / mai mare sau egal
#  ca 0
#   (i.e. dupa test si inainte de salt se salveaza adresa instructiunii
#    urmatoare in registrul $ra ($31));
## efectueaza: daca rs < / >= 0
```

```
#              atunci $ra <- adr. instr. urm. si apoi salt la eticheta
## au format I cu:  |    1   |   rs  |0x10/0x11|          depl          |
#                   ---------------------------------------------------
#                     6 biti   5 biti  5 biti          16 biti
#  unde depl este numarul de instructiuni masina (word-uri) peste care se
#   sare, numarand inclusiv instructiunea beq/bne curenta - acest numar este
#   determinat de compilator;
## exemplu:
#     lui $t0,1
#     bgezal $t0, et  # salveaza in $ra adr. instr. urm.: "lui $t0,2"
#     lui $t0,2
#     et:
#     lui $t1,3
#     jr $ra          # salt la adr. din $ra, adica la instr. "lui $t0,2"
## sunt utile la implementarea subrutinelor (a se vedea lectia 3);
```

Notam ca numarul de instructiuni peste care se sare la instructiunile de ramificare trebuie sa fie in intervalul -2^15, ..., 2^15-1, pentru a incapea pe 16 biti.

In procesoarele MIPS reale instructiunile de ramificare sunt ramificari intarziate (delayed branch), adica nu efectueaza saltul decat dupa ce s-a executat instructiunea urmatoare celei de ramificare ("delay slot"-ul ei). Ramificarile intarziate afecteaza calcularea offset-ului, deoarece acesta trebuie calculat in raport cu adresa instructiunea "delay slot" (PC+4). PCSpim nu simuleaza acest "delay slot" decat daca in meniul Simulator -> Settings bifam una din optiunile Bare machine sau Delayed Branches.

slt/sltu rd, rs, rt

```
## stabileste la mai mic strict cu/fara semn;
## efectueaza: daca rs < rt
#              atunci rd <- 1
#              altfel rd <- 0
#  deci rezultatul comparatiei nu provoaca salturi ci este stocat sub forma
#   valorii 1 (true) sau 0 (false) intr-un registru; acesta poate fi
#   consultat ulterior;
## o aplicatie este detectarea carry-ului la operatii aritmetice - a se vedea
#   exemplul referitor la adunarea adunarea a doua numere naturale lungi
#   (multi word) de mai jos;
#  o alta aplicatie este evaluarea expresiilor booleene: rezultatul
#   true/false al expresiilor elementare de tip comparatie este stocat
#   intr-un registru si poate fi usor compus logic cu alte valori de
#   adevar - a se vedea exemplul referitor la evaluarea expresiilor booleene
#   de mai jos;
## au format R cu:   |   0    |   rs  |   rt  |   rd  |   0   |0x2a/0x2b|
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti
```

slti/sltiu rt, rs, imm

```
## stabileste la mai mic strict imediat cu/fara semn;
## efectueaza: daca rs < imm
#              atunci rt <- 1
#              altfel rt <- 0;
## imm trebuie sa fie din intervalul -2^15, ..., 2^15-1, altfel se genereaza
#   eroare la compilare;
## au  format I cu:  |0xa/0xb |   rs  |   rt  |          imm           |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
## practic am constatat ca imm este stocat pe ultimii 16 biti ai
#   instructiunilor in format intern de intreg (cu semn); aceasta
#   configuratie este apoi interpretata cu/fara semn in functie de tipul
```

```
#   comparatiei (slti/sltiu);
#  de exemplu:
#     li $t1,0
#     sltiu $t2, $t1, -1  # $t0 devine 1
#  intr-adevar, -1 este stocat in ultimii 16 biti ca 0xffff, apoi sltu
#   interpreteaza aceasta config. fara semn, deci ca fiind nr. 32767, care,
#   evident, este > 0;

seq/sne/sle/sleu/sgt/sgtu/sge/sgeu rdest, rsrc1, rsrc2

## stabileste la
#   egal / diferit /
#   mai mic sau egal cu semn / mai mic sau egal fara semn /
#   mai mare strict cu semn / mai mare strict fara semn
#   mai mare sau egal cu semn / mai mare sau egal fara semn;
## efectueaza:
#   daca
#      rsrc1
#            = / != /
#            <= cu semn / <= fara semn /
#            > cu semn / > fara semn /
#            >= cu semn / >= fara semn
#      rsrc2
#   atunci rdest <- 1
#   altfel rdest <- 0
## sunt pseudoinstructiuni;

j eticheta

## salt;
## efectueaza: salt (neconditionat) la eticheta;
## are format J cu:  |  0x2   |             obiectiv                    |
#                    ---------------------------------------------------
#                      6 biti                26 biti
#  unde obiectiv este numarul de ordine absolut (nu deplasamentul fata de
#   instructiunea curenta) al instructiunii masina (word-ului) avand adresa
#   data de eticheta (deci instructiunea la care se sare) - acest numar este
#   determinat de compilator;
#  de exemplu la:
#    j et
#    lui $t0,1
#    et:
#    lui $t0,2
#  daca instructiunea (word-ul) la care se sare, "lui $t0,2", are adresa
#   0x0040002c, adica este al 0x0010000b-lea word din memorie, in codificarea
#   lui "j et" obiectiv va fi 0x010000b;
## intrucat obiectiv este pe 26 biti, cu "j" putem face salturi intr-o zona
#   de memorie de 2^26 word-uri;

jr rs

## salt la registru;
## efectueaza: salt la adresa din rs;
## format:           |   0    |   rs  |           0           |    8   |
#                    ---------------------------------------------------
#                      6 biti   5 biti          15 biti          6 biti

jal eticheta

## salt si legatura;
#   (i.e. inainte de salt se salveaza adresa instructiunii urmatoare in
#   registrul $ra ($31));
## efectueaza: $ra <- adr. instr. urm. si apoi salt la eticheta;
## are format J cu:  |  0x3   |             obiectiv                    |
```

```
#                    ------------------------------------------------
#                      6 biti                26 biti
#  unde obiectiv este numarul de ordine absolut (nu deplasamentul fata de
#   instructiunea curenta) al instructiunii masina (word-ului) avand adresa
#   data de eticheta (deci instructiunea la care se sare) - acest numar este
#   determinat de compilator;
## intrucat obiectiv este pe 26 biti, cu "jal" putem face salturi intr-o zona
#   de memorie de 2^26 word-uri;
## exemplu:
#     lui $t0,1
#     jal et         # salveaza in $ra adr. instr. urm.: "lui $t0,2"
#     lui $t0,2
#     et:
#     lui $t1,3
#     jr $ra         # salt la adr. din $ra, adica la instr. "lui $t0,2"
## este utila la implementarea subrutinelor (a se vedea lectia 3);

jalr rd, rs

## salt si legatura in registru;
## efectueaza: rd <- adr. instr. urm. si apoi salt la adr. din rs;
#  rd poate lipsi si atunci se considera $ra ($31);
## au format R cu:   |   0    |   rs  |   0   |   rd  |   0   |    9   |
#                    ------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti
## exemplu:
#     la $t0,et1    # $t0 <- adr. instr. "jr $t1"
#     jalr $t1,$t0  # $t1 <- adr. instr. "lui $t7,1", apoi salt la "jr $t1"
#     et2:
#     lui $t7,1
#     et1:
#     jr $t1        # salt la "lui $t7,1"
## este utila la implementarea subrutinelor (a se vedea lectia 3);

Exemple: simulare if, while, do, for, switch (cu jr):
~~~~~~~~

Aratam cum se translateaza cateva fragmente de cod in limbajul C, care
 contin structuri de control:

structura if-then-else:

if(x[i] == y)
 {z = 1; t = 2;}
else
 z = 10;

se translateaza in:

lw $t0,i
sll $t0,$t0,2 # shift. la stanga cu 2 echivaleaza cu inmult. cu 2^2=4
lw $t0,x($t0)
lw $t1,y
beq $t0,$t1 et1
  li $t0,10 # ramura else
  sw $t0,z
  b et2     # sar peste ramura then
et1:
  li $t0,1  # ramura then
  sw $t0,z
  li $t0,2
  sw $t0,t
et2:
```

structura if-then:

```
if(x[i] == y)
 {z = 1; t = 2;}
```

se translateaza in:

```
lw $t0,i
sll $t0,$t0,2
lw $t0,x($t0)
lw $t1,y
bne $t0,$t1,et  # testam de fapt negarea conditiei
  li $t0,1  # ramura then
  sw $t0,z
  li $t0,2
  sw $t0,t
et:
```

structura while:

```
while(x <= y){
  z=z+x;
  ++x;
}
```

se translateaza in:

```
et1:
lw $t0,x
lw $t1,y
bgt $t0,$t1,et2 # testam de fapt negarea conditiei
  lw $t0,z
  lw $t1,x
  add $t0,$t0,$t1
  sw $t0,z
  lw $t0,x
  addi $t0,$t0,1
  sw $t0,x
b et1
et2:
```

structura do:

```
do{
  z=z+x;
  ++x;
}while(x <= y)
```

se translateaza in:

```
et:
  lw $t0,z
  lw $t1,x
  add $t0,$t0,$t1
  sw $t0,z
  lw $t0,x
  addi $t0,$t0,1
  sw $t0,x
lw $t0,x
lw $t1,y
ble $t0,$t1,et
```

structura for:

```
for(i=0; i<n; ++i){
  z=z+x[i];
  y=y+i;
}
```

se translateaza in:

```
li $t0,0
sw $t0,i         # i=0;
et1:
lw $t0,i
lw $t1,n
bge $t0,$t1,et2  # testul i<n (testam de fapt i>=n)
  lw $t0,z
  lw $t1,i
  sll $t1,$t1,2
  lw $t1,x($t1)
  add $t0,$t0,$t1
  sw $t0,z
  lw $t0,y
  lw $t1,i
  add $t0,$t0,$t1
  sw $t0,y
lw $t0,i
addi $t0,$t0,1
sw $t0,i         # ++i
b et1
et2:
```

structura switch (se poate simula prin generalizarea lui if-then-else, dar
 vom arata o alta varianta):

```
switch(k){
 case 1: x=x+2; break;
 case 2: x=x+y; ++y; break;
 case 3: ++x; break;
 default: x=0;
}
```

se translateaza in:

```
.data
etichete: .word et1, et2, et3  # vector cu adresele ramurilor switch
.text
lw $t0,k
li $t1,1
blt $t0,$t1,etd # daca k<1 execut default
li $t1,3
bgt $t0,$t1,etd # daca k>3 execut default
li $t1,1
subu $t0,$t0,$t1  # translatez k a.i. sa fie indici 0, .., 2 in vt."etichete"
sll $t0,$t0,2
lw $t0,etichete($t0)  # acum $t0 contine adr. coresp. et1, et2 sau et3
jr $t0
et1:
  lw $t0,x
  addi $t0,$t0,2
  sw $t0,x
b ete # ies din switch (break)
et2:
  lw $t0,x
  lw $t1,y
  add $t0,$t0,$t1
  sw $t0,x
```

```
  lw $t0,y
  addi $t0,$t0,1
  sw $t0,y
b ete # ies din switch (break)
et3:
  lw $t0,x
  addi $t0,$t0,1
  sw $t0,x
b ete # ies din switch (break)
etd: # ramura default
 li $t0,0
 sw $t0,x
ete:
```

Observatii:
- daca o ramura nu are break, din ea va lipsi "b ete"; daca nu avem ramura default, vor lipsi liniile "etd:", "li $t0,0", "sw $t0,x" iar ramificarile la "etd" vor fi facute la "ete";
- modul de simulare al structurii switch de mai sus (bazat pe metoda salturilor indirecte - a se vedea lectia 3) are dezavantajul ca valorile din lista switch trebuie sa fie succesive;
  in schimb, simularea structurii switch cu ramificari imbricate (care generalizeaza ce am facut la if-then-else) are dezavantajul ca se pot face multe comparatii inainte de a se decide care ramura trebuie executata.

In exemplele de mai sus codul MIPS echivalent se poate face mai eficient daca nu mai (re)incarcam variabilele x, y, z din memorie in regsitri de fiecare data.

* Instructiuni de shiftare (deplasare) si rotire:

sll/srl/sra rd, rt, imm

```
## shiftare logica la stanga/logica la dreapta/aritmetica la dreapta
## efectueaza: rd <- rt << imm  (logic)
#   respectiv: rd <- rt >> imm  (logic)
#   respectiv: rd <- rt >> imm  (aritmetic)
#  imm trebuie sa fie din intervalul 0, ..., 31, altfel se genereaza eroare
#   la compilare;
## mai exact:
#   la sll: se deplaseaza bitii la stanga cu imm pozitii,
#           bitii care ies din word prin stanga se pierd,
#           locurile goale ramase in dreapta se umplu cu 0;
#   la srl: se deplaseaza bitii la dreapta cu imm pozitii,
#           bitii care ies din word prin dreapta se pierd,
#           locurile goale ramase in stanga se umplu cu 0;
#   la sra: ca la srl, dar locurile goale ramase in stanga
#           se umplu cu copii ale bitului de semn b31;
## au format R cu:   |   0    |   rs  |   rt  |   rd  |  imm  |  0/2/3 |
#                   ---------------------------------------------------
#                     6 biti   5 biti  5 biti  5 biti  5 biti  6 biti
## in acest caz campul rs este ignorat (practic am constatat ca se consid. 0);
```

sllv/srlv/srav rd, rt, rs

```
## shiftare logica variabila la stanga/logica variabila la dreapta/aritmetica
#   variabila la dreapta
## efectueaza: rd <- rt << rs  (logic)
#   respectiv: rd <- rt >> rs  (logic)
#   respectiv: rd <- rt >> rs  (aritmetic)
#  am constatat ca daca valoarea lui rs este in afara intervalului 0, ..., 31
#   se shifteaza cu rs mod 32 (in sensul restului pozitiv, de exemplu
#   1 mod 32 =1, -1 mod 32 = 31);
## (ce inseamna shiftare aritmetica sau logica - a se vedea mai sus)
```

```
## au format R cu:   |   0    |   rs  |   rt  |   rd  |   0  |  4/6/7 |
#                    ------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

Shiftarile la stanga/dreapta echivaleaza cu inmultiri/impartiri cu puteri ale
 lui 2. Mai exact:
 - shiftarea la stanga cu n echivaleaza cu inmultirea cu 2^n;
 - shiftarea aritmetica la dreapta cu n echivaleaza cu impartirea la 2^n;
   daca operandul ce trebuie impartit este pozitiv, se poate folosi si
   shiftarea logica la dreapta.
Astfel putem face inmultiri/impartiri mai rapide.

rol/ror rdest, rsrc1, rsrc2

## rotire la stanga/dreapta;
## efectueaza: pune in rdest configuratia de biti din rsrc1 deplasata la
#   stanga/dreapta cu rsrc2, a.i. bitii ce ies din word prin stanga/dreapta
#   sunt introdusi in aceeasi ordine in locul gol creat in dreapta/stanga;
## sunt pseudoinstructiuni care se translateaza in:
#    subu $1, $0, rsrc2
#    srlv/sllv $1, rsrc1, $1
#    sllv/srlv rdest, rsrc1, rsrc2
#    or rdest, rdest, $1
#  adica:
#    $at <- 0 - rsrc2
#    $at <- rsrc1 >>/<< $at
#      # practic se sfifteaza in sens contrar cu 32 - rsrc2, a.i. in $at
#      # ajunge exact ceea ce ar disparea din word daca in loc de rotire
#      # s-ar face shiftare logica obisnuita (in acelasi sens);
#    rdest <- rsrc1 <</>> rsrc2
#      # se shifteaza logic cu rsrc2 (in acelasi sens ca rotirea);
#      # 32 - rsrc2 biti ies din word, iar in partea opusa apare un gol de
#      # 32 - rsrc2 biti 0 (shiftarea e logica);
#    rdest <- rdest | $at
#      # partea care a iesit din word (si care s-a recuperat anterior in $at)
#      # se scrie peste golul de 32 - rsrc2 biti 0;

Exemplu (se ruleaza pas cu pas, urmarindu-se registrii):
~~~~~~~

.data
.text
main:
li $t1,11         # pe biti $t1: 0000 0000 0000 0000 0000 0000 0000 1011
                  # adica 0x0000000b
sll $t2, $t1, 2   # $t2: 0000 0000 0000 0000 0000 0000 0010 1100 (0x0000002c)
sll $t2, $t1, 28  # $t2: 1011 0000 0000 0000 0000 0000 0000 0000 (0xb0000000)
sll $t2, $t1, 29  # $t2: 0110 0000 0000 0000 0000 0000 0000 0000 (0x60000000)
sll $t2, $t1, 31  # $t2: 1000 0000 0000 0000 0000 0000 0000 0000 (0x80000000)
li $t1,0x50000000 # pe biti $t1: 0101 0000 0000 0000 0000 0000 0000 0000
                  # bitul de semn b31 este 0
srl $t2, $t1, 2   # $t2: 0001 0100 0000 0000 0000 0000 0000 0000 (0x14000000)
sra $t2, $t1, 2   # $t2: 0001 0100 0000 0000 0000 0000 0000 0000 (0x14000000)
 # deci daca b31 este 0, rezultatul este acelasi la srl, sra
li $t1,0xb0000000 # pe biti $t1: 1011 0000 0000 0000 0000 0000 0000 0000
                  # bitul de semn b31 este 1
srl $t2, $t1, 2   # $t2: 0010 1100 0000 0000 0000 0000 0000 0000 (0x2c000000)
sra $t2, $t1, 2   # $t2: 1110 1100 0000 0000 0000 0000 0000 0000 (0xec000000)
 # deci daca b31 este 1, rezultatul difera la srl, sra
li $t1,11         # $t1: 0000 0000 0000 0000 0000 0000 0000 1011 (0x0000000b)
li $t0,31
sll $t2, $t1, $t0 # $t2: 1000 0000 0000 0000 0000 0000 0000 0000 (0x80000000)
li $t0,32
sll $t2, $t1, $t0 # $t2: 0000 0000 0000 0000 0000 0000 0000 1011 (0x0000000b)
```

```
li $t0,67   # 67 = 2 * 32 + 3
sll $t2, $t1, $t0 # $t2: 0000 0000 0000 0000 0000 0000 0101 1000 (0x00000058)
li $t0,-1   # -1 = (-1) * 32 + 31
sll $t2, $t1, $t0 # $t2: 1000 0000 0000 0000 0000 0000 0000 0000 (0x80000000)
li $t0,-32
sll $t2, $t1, $t0 # $t2: 0000 0000 0000 0000 0000 0000 0000 1011 (0x0000000b)
li $t0,-67  # -67 = (-3) * 32 + 29
sll $t2, $t1, $t0 # $t2: 0110 0000 0000 0000 0000 0000 0000 0000 (0x60000000)
 # constatam ca in cazul shiftarilor variabile nr. de pozitii cu care se
 #  shifteaza poate fi in afara intervalului 0, ..., 31, si se va shifta
 #  cu acest nr. modulo 32 (restul considerat din 0, ..., 31);
 # notam ca shiftarea cu 0, 32 sau -32 lasa numarul pe loc (la fel shiftarea
 #  cu orice multiplu intreg de 32);
li $t1,0xb0000000 # $t1: 1011 0000 0000 0000 0000 0000 0000 0000
li $t0,30
sra $t2, $t1, $t0 # $t2: 1111 1111 1111 1111 1111 1111 1111 1110 (0xfffffffe)
li $t0,31
sra $t2, $t1, $t0 # $t2: 1111 1111 1111 1111 1111 1111 1111 1111 (0xffffffff)
li $t0,32
sra $t2, $t1, $t0 # $t2: 1011 0000 0000 0000 0000 0000 0000 0000 (0xb0000000)
li $t0,67   # 67 = 2 * 32 + 3
sra $t2, $t1, $t0 # $t2: 1111 0110 0000 0000 0000 0000 0000 0000 (0xf6000000)
li $t0,-1   # -1 = (-1) * 32 + 31
sra $t2, $t1, $t0 # $t2: 1111 1111 1111 1111 1111 1111 1111 1111 (0xffffffff)
li $t0,-32
sra $t2, $t1, $t0 # $t2: 1011 0000 0000 0000 0000 0000 0000 0000 (0xb0000000)
li $t0,-67  # -67 = (-3) * 32 + 29
sra $t2, $t1, $t0 # $t2: 1111 1111 1111 1111 1111 1111 1111 1101 (0xfffffffd)
 # deci in cazul shiftarii aritmetice variabile cu o valoare in afara
 #  intervalului 0, ..., 31 comportamentul este similar, dar tinem cont
 #  ca acum spatiul gol creat in stanga se umple cu bitul de semn;
li $t1,0x0000000b # $t1: 0000 0000 0000 0000 0000 0000 0000 1011
li $t0,29
rol $t2, $t1, $t0 # $t2: 0110 0000 0000 0000 0000 0000 0000 0001 (0x60000001)
li $t0,30
rol $t2, $t1, $t0 # $t2: 1100 0000 0000 0000 0000 0000 0000 0010 (0xc0000002)
li $t0,32
rol $t2, $t1, $t0 # $t2: 0000 0000 0000 0000 0000 0000 0000 1011 (0x0000000b)
li $t0,67   # 67 = 2 * 32 + 3
rol $t2, $t1, $t0 # $t2: 0000 0000 0000 0000 0000 0000 0101 1000 (0x00000058)
li $t0,-1   # -1 = (-1) * 32 + 31
rol $t2, $t1, $t0 # $t2: 1000 0000 0000 0000 0000 0000 0000 0101 (0x80000005)
li $t0,-67  # -67 = (-3) * 32 + 29
rol $t2, $t1, $t0 # $t2: 0110 0000 0000 0000 0000 0000 0000 0001 (0x60000001)
li $t1,0xb0000000 # $t1: 1011 0000 0000 0000 0000 0000 0000 0000
li $t0,30
ror $t2, $t1, $t0 # $t2: 1100 0000 0000 0000 0000 0000 0000 0010 (0xc0000002)
li $t0,31
ror $t2, $t1, $t0 # $t2: 0110 0000 0000 0000 0000 0000 0000 0001 (0x60000001)
li $t0,32
ror $t2, $t1, $t0 # $t2: 1011 0000 0000 0000 0000 0000 0000 0000 (0xb0000000)
li $t0,67   # 67 = 2 * 32 + 3
ror $t2, $t1, $t0 # $t2: 0001 0110 0000 0000 0000 0000 0000 0000 (0x16000000)
li $t0,-1   # -1 = (-1) * 32 + 31
ror $t2, $t1, $t0 # $t2: 0110 0000 0000 0000 0000 0000 0000 0001 (0x60000001)
li $t0,-32
ror $t2, $t1, $t0 # $t2: 1011 0000 0000 0000 0000 0000 0000 0000 (0xb0000000)
li $t0,-67  # -67 = (-3) * 32 + 29
ror $t2, $t1, $t0 # $t2: 1000 0000 0000 0000 0000 0000 0000 0101 (0x80000005)
 # deci comportamentul la rotiri este ca la shiftarile logice, dar bitii ce
 #  ies din word printr-o parte intra in el in aceeasi ordine prin cealalta
 #  parte; in particular notam ca rotirile cu multipli intregi de 32 lasa
 #  numarul pe loc;
li $v0,10
```

syscall
##########

* Instructiuni logice:

 Efectueaza operatii logice bit cu bit asupra unor word-uri.
 Tablele operatiilor pe biti:

x	y	x and y	x or y	x xor y	x nand y	x nor y
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	1	1	0	0	0

x	not x
0	1
1	0

not rdest, rsrc

not;
efectueaza: rdest <- not rsrc (operatia ~ din limbajul C);
este o pseudoinstructiune; am observat ca se transcrie prin:
nor rdest, rsrc, $0

and rd, rs, rt

and;
efectueaza: rd <- rs and rt (operatia & din limbajul C);
are format R cu: | 0 | rs | rt | rd | 0 | 0x24 |
---
6 biti   5 biti 5 biti 5 biti 5 biti 6 biti

andi rt, rs, imm

and imediat;
efectueaza: rt <- rs and imm (operatia & din limbajul C);
imm trebuie sa fie numar natural din intervalul 0, ..., 2^16-1
(altfel se raporteaza eroare la compilare);
are format I cu: | 0xc | rs | rt | imm |
---
6 biti   5 biti 5 biti   16 biti

or rd, rs, rt

or;
efectueaza: rd <- rs or rt (operatia | din limbajul C);
are format R cu: | 0 | rs | rt | rd | 0 | 0x25 |
---
6 biti   5 biti 5 biti 5 biti 5 biti 6 biti

ori rt, rs, imm

or imediat;
efectueaza: rt <- rs or imm (operatia | din limbajul C);
imm trebuie sa fie numar natural din intervalul 0, ..., 2^16-1
(altfel se raporteaza eroare la compilare);
are format I cu: | 0xd | rs | rt | imm |
---
6 biti   5 biti 5 biti   16 biti

xor rd, rs, rt

xor;
efectueaza: rd <- rs xor rt (operatia ^ din limbajul C);
are format R cu: | 0 | rs | rt | rd | 0 | 0x26 |

```
#                     6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

xori rt, rs, imm

## xor imediat;
## efectueaza: rt <- rs xor imm  (operatia ^ din limbajul C);
#   imm trebuie sa fie numar natural din intervalul 0, ..., 2^16-1
#   (altfel se raporteaza eroare la compilare);
## are format I cu:  |   0xe  |   rs  |   rt  |        imm             |
#                    ---------------------------------------------------
#                     6 biti   5 biti  5 biti         16 biti

nor rd, rs, rt
## nor;
## efectueaza: rd <- rs nor rt (= not (rs or rd) )
## are format R cu:  |   0    |   rs  |   rt  |   rd  |   0   |  0x27  |
#                    ---------------------------------------------------
#                     6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

 Instructiunile and, andi sunt folosite la anularea unor biti, iar or, ori
la setarea la 1 a unor biti. De asemenea, xor intre un regeistru si el insusi
il anuleaza (este o cale rapida de a-i anula continutul).

Exemplu (se ruleaza pas cu pas, urmarindu-se registrii):
~~~~~~~

.data
.text
main:
li $t0,0x00000030  # pe biti $t0: 0000 0000 0000 0000 0000 0000 0011 0000
li $t1,0x00000050  # pe biti $t1: 0000 0000 0000 0000 0000 0000 0101 0000

and $t2, $t0, $t1  # pe biti $t2: 0000 0000 0000 0000 0000 0000 0001 0000
                   #  adica 0x00000010
or  $t2, $t0, $t1  # pe biti $t2: 0000 0000 0000 0000 0000 0000 0111 0000
                   #  adica 0x00000070
xor $t2, $t0, $t1  # pe biti $t2: 0000 0000 0000 0000 0000 0000 0110 0000
                   #  adica 0x00000060
andi $t2, $t1, 176 # pe biti 176: 0000 0000 0000 0000 0000 0000 1011 0000
                   # rezulta $t2: 0000 0000 0000 0000 0000 0000 0001 0000
                   #    adica 0x00000010
ori  $t2, $t1, 176 # pe biti 176: 0000 0000 0000 0000 0000 0000 1011 0000
                   # rezulta $t2: 0000 0000 0000 0000 0000 0000 1111 0000
                   #    adica 0x000000f0
xori $t2, $t1, 176 # pe biti 176: 0000 0000 0000 0000 0000 0000 1011 0000
                   # rezulta $t2: 0000 0000 0000 0000 0000 0000 1110 0000
                   #    adica 0x000000e0
not $t2, $t1       # pe biti $t2: 1111 1111 1111 1111 1111 1111 1010 1111
                   #  adica 0xfffffffaf
nor $t2, $t1, $t0  # pe biti $t2: 1111 1111 1111 1111 1111 1111 1000 1111
                   #  adica 0xffffff8f
## aratam cum putem folosi or, ori la setarea la 1 a unor biti;
## de exemplu vrem sa setam bitii b4, b5 din $t1;
##  atunci facem ori intre $t1 si un halfword care are toti bitii 0 in afara
##  afara de b4 si b5 unde are 1: 0000 0000 0011 0000 adica 0x0030
ori $t2, $t1, 0x0030       # $t2: 0000 0000 0000 0000 0000 0000 0111 0000
                           # adica 0x00000070
 # metoda nu merge decat daca pozitiile sunt de la 0 la 15;
 # pentru a seta biti de pe pozitii mai mari (pana la 32) sau in cazul cand
 #  pozitiile sunt determinate din calcule, putem proceda astfel:
li $s0, 4                  #
li $s1, 17                 # pozitiile unde setam: 4, 17
li $s2, 0x1
sll $s2, $s2, $s0          # $s2: 0000 0000 0000 0000 0000 0000 0001 0000
```

```
                            #  are setat b4
move $s3, $s2               # $s3: 0000 0000 0000 0000 0000 0000 0001 0000
li $s2, 0x1
sll $s2, $s2, $s1           # $s2: 0000 0000 0000 0010 0000 0000 0000 0000
                            #  are setat b17
or $s3, $s3, $s2            # $s3: 0000 0000 0000 0010 0000 0000 0001 0000
                            #  deci are setati exacti bitii ce trebuie setati
                            #  in $t1
or  $t2, $t1, $s3           # $t2: 0000 0000 0000 0010 0000 0000 0101 0000
                            # adica 0x00020050
## aratam cum putem folosi and, andi la anularea unor biti;
## de exemplu vrem sa anulam bitii b4, b5 din $t1;
##  atunci facem andi intre $t1 si un halfword care are toti bitii 1 in afara
##  afara de b4 si b5 unde are 0: 1111 1111 1100 1111 adica 0xffcf
andi $t2, $t1, 0xffcf       # $t2: 0000 0000 0000 0000 0000 0000 0100 0000
                            # adica 0x00000040
 # metoda nu merge decat daca pozitiile sunt de la 0 la 15;
 # pentru a seta biti de pe pozitii mai mari (pana la 32) sau in cazul cand
 #  pozitiile sunt determinate din calcule, putem proceda astfel (similar
 #  ca la setare, doar ca masca se neaga in prealabil si se face and, nu or):
li $s0, 4                   #
li $s1, 17                  # pozitiile unde anulam: 4, 17
li $s2, 0x1
sll $s2, $s2, $s0           # $s2: 0000 0000 0000 0000 0000 0000 0001 0000
                            #  are setat b4
move $s3, $s2               # $s3: 0000 0000 0000 0000 0000 0000 0001 0000
li $s2, 0x1
sll $s2, $s2, $s1           # $s2: 0000 0000 0000 0010 0000 0000 0000 0000
                            #  are setat b17
or $s3, $s3, $s2            # $s3: 0000 0000 0000 0010 0000 0000 0001 0000
                            #  deci are setati exacti bitii ce trebuie anulati
                            #  in $t1
not $s3, $s3                # $s3: 1111 1111 1111 1101 1111 1111 1110 1111
                            #  deci este exact masca cu care trebuie facut and
and  $t2, $t1, $s3          # $t2: 0000 0000 0000 0000 0000 0000 0100 0000
                            # adica 0x00000040
li $v0,10
syscall
##########

Exemplu: evaluarea expresiilor booleene, folosind instructiunile de
~~~~~~~~  comparatie (slt, seq, etc.):

Dorim sa evaluam expresia booleana: (x<=y || x!=z ) && (y == z)
 pentru niste numere x, y, z oarecare.

.data
x: .word 1
y: .word 2
z: .word 3
e: .space 4
.text
main:
 # evaluam x <= y si stocam raspunsul 1(true)/0(false) in $s0
lw $t0,x
lw $t1,y
sleu $s0,$t0,$t1
 # evaluam x != z si stocam raspunsul 1/0 in $s1
lw $t0,x
lw $t1,z
sne $s1,$t0,$t1
 # evaluam (x<=y || x!=z ), adica ($s0 or $s1),si stocam raspunsul 1/0 in $s0
or $s0,$s0,$s1
 # evaluam y == z si stocam raspunsul 1/0 in $s1
```

```
lw $t0,y
lw $t1,z
seq $s1,$t0,$t1
 # evaluam (x<=y || x!=z ) && (y == z), adica ($s0 and $s1),
 #  si stocam raspunsul 1/0 in $v0, apoi in memorie
and  $v0,$s0,$s1
sw $v0,e
li $v0,10
syscall
##########

Comentarii:
- daca in loc de instructiunile de comparatie slt, seq, ... le foloseam pe
 cele de ramificare blt, beq, ..., programul era mai complicat (continea
 multe ramificari) iar structura lui de ramuri depindea puternic de structura
 expresiei si era mai greu de generalizat;
- desi "or" si "and" lucreaza pe biti, operanzii lor de aici au toti bitii 0
 in afara eventual de b0 - astfel putem considera ca ele implementeaza "sau"
 si "si" logic (nu pe biti) - analogul operatiilor "||", "&&" din limbajul C;
- programul se poate face mai eficient daca nu mai (re)incarcam x, y, z din
 memorie in regsitri de fiecare data.

* Instructiuni aritmetice cu intregi:

Instructiunile "fara semn" presupun operanzii numere naturale din intervalul
 0, ..., 2^32-1 iar operatiile se fac in numere naturale.
Instructiunile "cu semn" presupun operanzii numere intregi din intervalul
 -2^31, ..., 2^31-1 iar operatiile se fac in numere intregi.
La instructiunile "cu depasire", daca rezultatul iese din intervalul
 considerat se genereaza o exceptie.
La instructiunile "fara depasire", daca rezultatul iese din intervalul
 considerat nu se genereaza o exceptie ci se trunchiaza; de exemplu in cazul
 "fara semn" se reduce modulo 2^32.

abs rdest, rsrc

## valoare absoluta;
## efectueaza: rdest <- |rsrc|
## este o pseudoinstructiune care se translateaza in:
#    addu rdest, $0, rsrc
#    bgez rsrc 8
#    sub rdest, $0, rsrc
#  adica:
#    rdest <- 0 + rsrc
#    daca rsrc >=0 sari peste instructiunea asta si peste cea urmatoare
#    rdest <- 0 - rsrc

neg/negu rdest, rsrc

## opusul cu/fara depasire
## efectueaza: rdest <- - rsrc
## sunt pseudoinstructiuni;
#  practic am constatat ca ele se transcriu prin: sub/subu rdest, $0, rsrc
#   (iar comportamentul legat de semn si depasire rezulta din cel al lui
#   sub/subu)

add rd, rs, rt

## adunare cu semn cu depasire;
## efectueaza: rd <- rs + rt
## are format R cu:  |   0    |   rs  |   rt  |   rd  |   0   |  0x20  |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti
```

```
addu rd, rs, rt

## adunare fara semn fara depasire;
## efectueaza: rd <- rs + rt
## are format R cu:  |   0    |   rs  |   rt  |   rd  |   0   |  0x21  |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

addi rt, rs, imm

## adunare imediata cu semn cu depasire;
## efectueaza: rt <- rs + imm
#  imm poate fi si negativ si se considera cu semn extins la 32 biti;
## are format I cu:  |   0x8  |   rs  |   rt  |        imm             |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti         16 biti

addiu rt, rs, imm

## adunare imediata fara semn fara depasire;
## efectueaza: rt <- rs + imm
#  imm poate fi si negativ si se considera cu semn extins la 32 biti;
## are format I cu:  |   0x9  |   rs  |   rt  |        imm             |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti         16 biti

add/addu rdest, rsrc, imm

## adunare imediata cu/fara semn, depasire;
## efectueaza: rdest <- rsrc + imm
## sunt pseudoinstructiuni;
#  practic am constatat ca ele se transcriu prin:
#   addi/addiu rdest, rsrc, imm
#   (iar comportamentul legat de semn si depasire rezulta din cel al lui
#   addi/addiu)

add/addu reg, imm

## adunare imediata cu/fara semn, depasire;
## efectueaza: reg <- reg + imm
## sunt pseudoinstructiuni;
#  practic am constatat ca ele se transcriu prin:
#   addi/addiu reg, reg, imm
#   (iar comportamentul legat de semn si depasire rezulta din cel al lui
#   addi/addiu)

sub rd, rs, rt

## scadere cu semn cu depasire;
## efectueaza: rd <- rs - rt
## are format R cu:  |   0    |   rs  |   rt  |   rd  |   0   |  0x22  |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

subu rd, rs, rt

## scadere fara semn fara depasire;
## efectueaza: rd <- rs - rt
## are format R cu:  |   0    |   rs  |   rt  |   rd  |   0   |  0x23  |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

sub/subu rdest, rsrc, imm
```

```
## scadere imediata cu/fara semn, depasire;
## efectueaza: rdest <- rsrc - imm
## sunt pseudoinstructiuni;
#  practic am constatat ca ele se transcriu prin:
#   addi/addiu rdest, rsrc, -imm
#   (iar comportamentul legat de semn si depasire rezulta din cel al lui
#   addi/addiu)

sub/subu reg, imm

## scadere imediata cu/fara semn, depasire;
## efectueaza: reg <- reg - imm
## sunt pseudoinstructiuni;
#  practic am constatat ca ele se transcriu prin:
#   addi/addiu reg, reg, -imm
#   (iar comportamentul legat de semn si depasire rezulta din cel al lui
#   addi/addiu)

Exemplu (se ruleaza pas cu pas, urmarind registrii $t0, $t1, $t2):
~~~~~~~

.text
main:
li $t0,0         # $t0=0x00000000
addi $t0,$t0,-1  # $t0=0xffffffff; obs. cum s-a propagat semnul
li $t0,0         # $t0=0x00000000
addiu $t0,$t0,-1 # $t0=0xffffffff; obs. ca da la fel
##########
li $t0,0x80000000  # $t0=0x80000000=-2^31 (ca intreg e c.m.mic intreg <0)
# addi $t0,$t0,-1  # se genereaza excepie
li $t0,0x7fffffff  # $t0=0x7fffffff=2^31-1 (ca intreg e c.m.mare intreg >0)
# addi $t0,$t0,1   # se genereaza excepie
##########
li $t0,0           # $t0=0 (cel mai mic natural)
addiu $t0,$t0,-1   # $t0=0xffffffff=2^32-1, adica (0-1) mod 2^32
li $t0,0xffffffff  # $t0=2^32-1 (cel mai mare natural)
addiu $t0,$t0,1    # $t0=0, adica (2^32-1+1) mod 2^32
##########
li $t0,0x80000000  # $t0=0x80000000=-2^31 (ca intreg e c.m.mic intreg <0)
li $t1,-1          # $t1=0xffffffff=-1 (privit ca intreg)
# add $t2,$t0,$t1  # se genereaza excepie
li $t0,0x7fffffff  # $t0=0x7fffffff=2^31-1 (ca intreg e c.m.mare intreg >0)
li $t1,1           # $t1=0x00000001=1 (privit ca intreg)
# add $t0,$t0,$t1  # se genereaza excepie
##########
li $t0,0           # $t0=0 (cel mai mic natural)
li $t1,-1
addu $t2,$t0,$t1   # $t0=0xffffffff=2^32-1, adica (0-1) mod 2^32
li $t0,0xffffffff  # $t0=2^32-1 (cel mai mare natural)
li $t1,1
addu $t2,$t0,$t1   # $t0=0, adica (2^32-1+1) mod 2^32
##########
li $t0,0         # $t0=0x00000000
li $t1,1
sub $t2,$t0,$t1  # $t2=0xffffffff; obs. cum s-a propagat semnul
subu $t2,$t0,$t1 # $t2=0xffffffff; obs. ca da la fel
##########
li $t0,0x80000000  # $t0=0x80000000=-2^31 (ca intreg e c.m.mic intreg <0)
li $t1,1
# sub $t2,$t0,$t1  # se genereaza excepie
li $t0,0x7fffffff  # $t0=0x7fffffff=2^31-1 (ca intreg e c.m.mare intreg >0)
li $t1,-1
# sub $t0,$t0,$t1  # se genereaza excepie
##########
```

```
li $t0,0           # $t0=0 (cel mai mic natural)
li $t1,1
subu $t2,$t0,$t1   # $t0=0xffffffff=2^32-1, adica (0-1) mod 2^32
li $t0,0xffffffff  # $t0=2^32-1 (cel mai mare natural)
li $t1,-1
subu $t2,$t0,$t1   # $t0=0, adica (2^32-1-(-1)) mod 2^32
##########
li $v0,10
syscall
##########
```

Obs: indiferent daca aplicam add/addi sau addu/addiu, configuratia pe biti rezultata este aceeasi; faptul ca reprezinta un natural 0,...,2^32-1 sau un intreg -2^31,...,2^31-1 si eventuala generare a unei exceptii depinde de cum interpretam ulterior aceasta configuratie.

Exemplu: Algoritmul lui Euclid cu scaderi (se ruleaza pas cu pas, urmarind registrii $t0, $t1, $t2);

Translatam urmatorul program C:

```
t0=4; t1=6;
while(t0!=t1)
  if(t0>t1)t0-=t1;
   else t1-=t0;
t2=t0;
```

```
.text
main:
  li $t0,4
  li $t1,6
 inceput:
  beq $t0,$t1,egale
    bltu $t0,$t1,maimic
      subu $t0,$t0,$t1
      b inceput
    maimic:
      subu $t1,$t1,$t0
      b inceput
  egale:
    move $t2,$t0
li $v0,10
syscall
##########
```

```
mult rs, rt

## inmultire cu semn (locatia dest. e suf. de mare ca sa nu apara depasire);
## efectueaza: lo <- word-ul low din rs * rt; hi <- word-ul hi din rs * rt;
#     (inmultire cu semn)
#  valorile se pot recupera din reg. lo, hi cu instructiunile mflo, mfhi;
## are format R cu:  |   0    |   rs  |   rt  |   0  |   0   |  0x18  |
#                    -------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

multu rs, rt

## inmultire fara semn (locatia dest. e suf.de mare ca sa nu apara depasire);
## efectueaza: lo <- word-ul low din rs * rt; hi <- word-ul hi din rs * rt;
#     (inmultire fara semn)
#  valorile se pot recupera din reg. lo, hi cu instructiunile mflo, mfhi;
## are format R cu:  |   0    |   rs  |   rt  |   0   |   0   |  0x19  |
#                    -------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti
```

```
mul rdest, rsrc1, rsrc2

## inmultire fara depasire;
## efectueaza: rdest <- rsrc1 * rsrc2;
## este o pseudoinstructiune;

mulo rdest, rsrc1, rsrc2

## inmultire cu semn cu depasire;
## efectueaza: rdest <- rsrc1 * rsrc2
#    generand exceptie daca rezultatul nu este in intervalul de nr. intregi
#    (cu semn) reprezentabil pe 1 word, adica -2^31, ..., 2^31-1
## este o pseudoinstructiune care se translateaza in:
#    mult rsrc1, rsrc2
#    mfhi $1
#    mflo rdest
#    sra rdest, rdest, 31
#    beq $1, rdest, 8
#    break $0
#    mflo rdest
#  adica:
#    (hi,lo) <- rsrc1 * rsrc2
#    $1 <- hi
#    rdest <- lo
# (acum daca produsul, ca numar intreg (deci cu semn din intervalul
# -2^31, ..., 2^31-1), a incaput in lo, hi este extensia bitului b31 din lo)
#    rdest <- rdest >> 31 (shift aritmetic)
# (rdest se shifteaza aritmetic la dreapta cu 31 pozitii, astfel ca acum
# contine peste tot copii ale bitului sau b31)
#    daca $1 egal cu rdest 0 (i.e. hi chiar este extensia bitului b31 din lo)
#      sari peste instructiunea asta si peste cea urmatoare  (adica la noul
#      "mflo")
#    genereaza exceptie
#    rdest <- lo
# (deci copiem din nou pe lo (care stim acum ca contine tot produsul
# rsrc1 * rsrc2) in rdest, care a fost alterat mai sus de shiftare)

mulou rdest, rsrc1, rsrc2

## inmultire fara semn cu depasire;
## efectueaza: rdest <- rsrc1 * rsrc2
#    generand exceptie daca rezultatul nu este in intervalul de nr. naturale
#    reprezentabil pe 1 word, adica 0, ..., 2^32-1
## este o pseudoinstructiune care se translateaza in:
#    multu rsrc1, rsrc2
#    mfhi $1
#    beq $1, $0, 8
#    break $0
#    mflo rdest
#  adica:
#    (hi,lo) <- rsrc1 * rsrc2
# (acum daca produsul, ca numar natural (deci din intervalul 0, ..., 2^32-1)
# a incaput in lo, hi este 0)
#    $1 <- hi
#    daca $1 este 0 (i.e. n-avem depasire iar produsul rsrc1 * rsrc2 a
#      incaput in lo), sari peste instructiunea asta si peste cea urmatoare
$      (adica la "mflo")
#    genereaza exceptie
#    rdest <- lo
# (deci copiem pe lo (care stim acum ca contine tot produsul rsrc1 * rsrc2)
# in rdest

div rs, rt
```

```
## impartire cu semn cu depasire;
## efectueaza: lo <- rs / rt; hi <- rs % rt;
#  daca un operand este negativ % este nespecificat de arhitectura MIPS si
#     depinde de conventiile masinii gazda;
#  valorile se pot recupera din reg. lo, hi cu instructiunile mflo, mfhi;
## are format R cu:  |   0    |   rs  |   rt  |   0   |   0   |  0x1a  |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

divu rs, rt

## impartire fara semn fara depasire;
## efectueaza: lo <- rs / rt; hi <- rs % rt;
#  daca un operand este negativ % este nespecificat de arhitectura MIPS si
#     depinde de conventiile masinii gazda;
#  valorile se pot recupera din reg. lo, hi cu instructiunile mflo, mfhi;
## are format R cu:  |   0    |   rs  |   rt  |   0   |   0   |  0x1b  |
#                    ---------------------------------------------------
#                      6 biti   5 biti  5 biti  5 biti  5 biti  6 biti

div rdest, rsrc1, rsrc2
divu rdest, rsrc1, rsrc2

## impartire cu/fara semn, depasire;
## efectueaza: rdest <- rsrc1 / rsrc2;
## sunt pseudoinstructiuni care se translateaza in:
#    bne rsrc2, $zero, 8
#    break $zero
#    div/divu rsrc1, rsrc2
#    mflo rdest
#  adica:
#    daca rsrc2 nu este 0, sari peste instructiunea asta si peste cea
#      urmatoare (adica la "div/divu")
#    genereaza exceptie
#    lo <- rsrc1 / rsrc2; hi <-rsrc1 % rsrc2 (operatii cu/fara semn,depasire)
#    rdest <- lo

rem/remu rdest, rsrc1, rsrc2

## rest
## efectueaza: rdest <- rsrc1 % rsrc2;
#  daca un operand este negativ % este nespecificat de arhitectura MIPS si
#     depinde de conventiile masinii gazda;
## sunt pseudoinstructiuni care se translateaza in:
#    bne rsrc2, $0, 4
#    break $0
#    div/divu rsrc1, rsrc2
#    mfhi rdest
#  adica:
#    daca rsrc2 nu este 0, sari peste instructiunea asta si peste cea
#      urmatoare (adica la "div/divu")
#    genereaza exceptie
#    lo <- rsrc1 / rsrc2; hi <-rsrc1 % rsrc2 (operatii cu/fara semn,depasire)
#    rdest <- hi
#  deci comportamentul legat de semn si depasire rezulta din cel al lui
#    div/divu;

Exemplu ce evidentiaza diverse proprietati legate de inmultiri/impartiri
~~~~~~~ (se ruleaza pas cu pas urmarindu-se continutul registrilor $t0, $t1,
         lo, hi):
.text
main:
li $t0,0x80000123
li $t1,0x00000002
```

```
multu $t0,$t1
# acum hi=0x00000001, lo=0x00000246
# intr-adevar, 0x80000123 * 2 = 0x0000000100000246 (inm. de nr. naturale)
# partea hi este semnificativa (i.e. nenula)
##########
li $t0,0x80000123
li $t1,0x00000001
multu $t0,$t1
# acum hi=0x00000000, lo=0x80000123
# intr-adevar, 0x80000123 * 1 = 0x0000000080000123 (inm. de nr. naturale)
# partea hi este nesemnificativa (i.e. nula)
##########
li $t0,0x80000000  # ca nr. intreg, $t0 = -2^31
li $t1,0x00000001  # ca nr. intreg, $t1 = 1
mult $t0,$t1
# acum hi=0xffffffff, lo=0x80000000
# intr-adevar, -2^31 * 1 = -2^31, care se repr. pe 2 word: 0xffffffff80000000
# partea hi este nesemnificativa (i.e. rezultatul a incaput in partea low),
#   dar la inmultirea cu semn se propaga bitul de semn, care aici e 1;
#   daca faceam "multu" rezulta hi=0x00000000, lo=0x80000000
##########
li $t0,0x80000000  # ca nr. intreg, $t0 = -2^31
li $t1,0x00000002  # ca nr. intreg, $t1 = 2
mult $t0,$t1
# acum hi=0xffffffff, lo=0x00000000
# intr-adevar, -2^31 * 2 = -2^32, care se repr. pe 2 word: 0xffffffff00000000
# partea hi este semnificativa (i.e. este parte a rezultatului, deoarece
#   rezultatul nu a incaput doar in partea low), deci 0xffffffff din hi
#   nu este extensia de semn a bitului b31 din lo; daca faceam "multu"
#   rezulta hi=0x00000001, lo=0x00000000 (adica 2^31 * 2 = 2^32)
##########
li $t0,0x78000000  # ca nr. intreg, $t0 = 2^30 + 2^29 + 2^28 + 2^27
li $t1,0x00000002  # ca nr. intreg, $t1 = 2
mult $t0,$t1
# acum hi=0x00000000, lo=0xf0000000
# intr-adevar, (2^30 + 2^29 + 2^28 + 2^27) * 2 = (2^31 + 2^30 + 2^29 + 2^28),
#   care se repr. pe 2 word: 0x00000000f0000000
# partea hi este semnificativa (i.e. parte a rezultatului, chiar daca e nula,
#   deoarece rezultatul este un nr. pozitiv care nu se poate reprezenta ca
#   INTREG (cu semn) doar in lo, intrucat nu este intre -2^31, ..., 2^31-1);
#   astfel, desi bitul b31 din lo este 1, el nu s-a propagat in hi
##########
li $t0,0x80000000  # ca nr. natural, $t0 = 2^31
li $t1,0x00000002  # ca nr. natural, $t1 = 2
### mulou $t2,$t0,$t1
# genereaza exceptie, deoarece prod. este 2^32 > 2^32-1, cel mai mare
#   nr. natural (deci fara semn) reprezentabil pe 1 word
##########
li $t0,0x40000000  # ca nr. intreg, $t0 = 2^30
li $t1,0x00000002  # ca nr. intreg, $t1 = 2
### mulo $t2,$t0,$t1
# genereaza exceptie, deoarece produsul este 2^31 > 2^31-1, cel mai mare
#   nr. inreg (deci cu semn) reprezentabil pe 1 word
##########
li $t0,0xbfffffff  # ca nr. intreg, $t0 = -2^30-1
li $t1,0x00000002  # ca nr. intreg, $t1 = 2
### mulo $t2,$t0,$t1
# genereaza exceptie, deoarece produsul este -2^31-2 < -2^31, cel mai mic
#   nr. inreg (deci cu semn) reprezentabil pe 1 word
##########
li $t0,0x00000041  # ca nr. intreg sau ca nr. natural, $t0=65
li $t1,0x00000002  # ca nr. intreg sau ca nr. natural, $t1=2
div $t0,$t1
# obtinem lo=0x00000020 (i.e. catul = 32), hi=0x00000001 (i.e. retsul = 1)
```

```
divu $t0,$t1
# obtinem lo=0x00000020 (i.e. catul = 32), hi=0x00000001 (i.e. retsul = 1)
##########
li $t0,0x00000041  # ca nr. intreg, $t0=65
li $t1,0xfffffffe  # ca nr. intreg, $t1=-2
div $t0,$t1
# obtinem lo=0xffffffe0 (i.e. catul = -32), hi=0x00000001 (i.e. retsul = 1)
##########
li $t0,0x00000041  # ca nr. natural, $t0=65
li $t1,0xfffffffe  # ca nr. natural, $t1=2^32-2
divu $t0,$t1
# obtinem lo=0x00000000 (i.e. catul = 0), hi=0x00000041 (i.e. retsul = 65)
##########
li $t0,0x80000000  # ca nr. intreg, $t0=-2^31
li $t1,0xffffffff  # ca nr. intreg, $t1=-1
div $t0,$t1
# ar trebui sa genereza exceptie, deoarece catul este 2^31 si nu incape ca
#    intreg cu semn in lo (ar trebui sa fie < 2^31-1);
# totusi nu se genereaza exceptie, lo,hi raman nemodificate si se merge mai
#    departe
##########
li $t0,0x80000000  # ca nr. intreg, $t0=-2^31
li $t1,0xffffffff  # ca nr. intreg, $t1=-1
divu $t0,$t1
# ca mai sus, nu se genereaza exceptie, lo,hi raman nemodificate si se merge
#    mai departe
##########
li $v0,10
syscall
##########
```

Exemplu: Algoritmul lui Euclid cu impartiri (se ruleaza pas cu pas, urmarind
~~~~~~~~ continutul registrilor $t0, $t1, $t2);
Translatam urmatorul program C:

```
t0=4; t1=6;
while(t1!=0){
  hi=t0 % t1;
  t0=t1; t1=hi;
}
t2=t0;
```

```
.text
main:
  li $t0,4
  li $t1,6
 inceput:
  beqz $t1, sfarsit
  divu $t0,$t1   # lo <- $t0 / $t1; hi <- $t0 % $t1
  move $t0,$t1   # $t0 <- $t1
  mfhi $t1       # $t1 <- hi
  b inceput
 sfarsit:
  move $t2,$t0
li $v0,10
syscall
##########
```

Exemplu: calculeaza maximul elementelor dintr-un vector de word, aratand
~~~~~~~~   diverse metode de aparcurge un vector (se ruleaza pas cu pas
           urmarind continutul memoriei si registrilor folositi);
Translatam urmatorul program C:

```
int v[]={2, 1, 3, 2, 3}, n=5, i, max;
```

```
max=v[0];
for(i=1; i<n; ++i)
  if(v[i]>max) max=v[i];

.data
 v:.word 2, 1, 3, 2, 3
 n:.word 5
 i:.space 4
 max:.space 4
.text
main:
#varianta ce traduce fidel programul C
  lw $t0,v
  sw $t0,max  # max=v[0]
  li $t0,1
  sw $t0,i             # initializare for: i=1
 continuare1:
  lw $t1,n
  bge $t0,$t1,iesire1  # testul de continuare din for: daca i>=n ies
  mulou $t0,$t0,4  # acum $t0 contine distanta in octeti a lui v[i] fata de v
  lw $t0,v($t0)    # acum $t0 contine v[i]
  lw $t1,max
  ble $t0,$t1,et1  # daca v[i]<=max nu actualizez max
  sw $t0,max       # actualizez max=v[i]
 et1:
  lw $t0,i
  addi $t0,$t0,1
  sw $t0,i             # reactualizare for: ++i
 b continuare1         # reiau ciclul for
 iesire1:
#varianta optimizata:
  lw $t2,v  # $t2 va retine max curent
  li $t0,1  # $t0 va retine indicele sau distanta elem. curent fata de v
  lw $t1,n  # $t1 va retine numarul de elemente
 continuare2:
  bge $t0,$t1,iesire2
  sll $t0,$t0,2  # shiftarea logica la stg. cu 2 inseamna inmultirea cu 2^2=4
  lw $t3,v($t0)
  srl $t0,$t0,2  # shiftarea logica la dr. cu 2 inseamna impartirea la 2^2=4
  ble $t3,$t2,et2
  move $t2,$t3
 et2:
  addi $t0,$t0,1
  b continuare2
 iesire2:
  sw $t2,max
li $v0,10
syscall
##########
```

Obs. ca varianta a doua este mai scurta si mai rapida, dar foloseste mai multi registri (si astfel avem mai putini registri disponibili pentru o prelucrare mai complexa la fiecare iteratie).

Exemplu: adunarea a doua matrici de numere naturale (stocate liniarizat):
~~~~~~~~
Translatam urmatorul program C:

```
int x[2][3]={{0x1,0x2,0x3},{0x4,0x5,0x6}},
    y[2][3]={{0x10,0x20,0x30},{0x40,0x50,0x60}},
    z[2][3],
    nl=2, nc=3, i, j;
for(i=0;i<nl;++i)
 for(j=0;j<nc;++j)
```
~~~~~~~~

```
  z[i][j]=x[i][j]+y[i][j];

.data
x: .word 0x1,0x2,0x3,0x4,0x5,0x6       # si in C si aici, matricile sunt
y: .word 0x10,0x20,0x30,0x40,0x50,0x60 #  stocate liniarizat (se stocheaza
z: .space 24                           #  liniile una dupa alta)
 # suma va fi 0x11,0x22,0x33,0x44,0x55,0x66
nl: .word 2
nc: .word 3
ii: .space 4  # nu pot declara variabila "j" caci e cuvant cheie
jj: .space 4  #  (mnemonicul instructiunii se salt) asa ca folosesc "ii","jj"
.text
main:
li $t0,0
sw $t0,ii # ii=0
intrare_ciclu_linii:
lw $t6,nl
bge $t0,$t6,iesire_ciclu_linii
  li $t1,0
  sw $t1,jj # jj=0
  intrare_ciclu_coloane:
  lw $t7,nc
  bge $t1,$t7,iesire_ciclu_coloane
    mulo $t2,$t0,$t7    # $t2=ii*nc+jj, indicele in vectorul de liniarizare
    add   $t2,$t2,$t1   #   corespunzator pozitiei [ii][jj] din matrici
    sll $t2,$t2,2       # acum $t2 este offsetul in octeti al elementului
    lw $t3,x($t2)    # $t3=x[ii][jj]
    lw $t4,y($t2)    # $t4=y[ii][jj]
    add $t3,$t3,$t4  # $t3=x[ii][jj]+y[ii][jj]
    sw $t3,z($t2)    # z[ii][jj]=x[ii][jj]+y[ii][jj]
  lw $t1,jj
  addi $t1,$t1,1
  sw $t1,jj       # ++jj
  b intrare_ciclu_coloane
  iesire_ciclu_coloane:
lw $t0,ii
addi $t0,$t0,1
sw $t0,ii       # ++ii
b intrare_ciclu_linii
iesire_ciclu_linii:
li $v0,10
syscall
##########
```

Mai eficient, programul se poate scrie:

```
.data
x: .word 0x1,0x2,0x3,0x4,0x5,0x6       # si in C si aici, matricile sunt
y: .word 0x10,0x20,0x30,0x40,0x50,0x60 #  stocate liniarizat (se stocheaza
z: .space 24                           #  liniile una dupa alta)
 # suma va fi 0x11,0x22,0x33,0x44,0x55,0x66
nl: .word 2
nc: .word 3
.text
main:
lw $t6,nl
lw $t7,nc
li $t0,0  # ii=0
intrare_ciclu_linii:
bge $t0,$t6,iesire_ciclu_linii
  li $t1,0  # jj=0
  intrare_ciclu_coloane:
  bge $t1,$t7,iesire_ciclu_coloane
    mulo $t2,$t0,$t7    # $t2=ii*nc+jj, indicele in vectorul de liniarizare
```

```
    add   $t2,$t2,$t1   #   corespunzator pozitiei [ii][jj] din matrici
    sll $t2,$t2,2       # acum $t2 este offsetul in octeti al elementului
    lw $t3,x($t2)    # $t3=x[ii][jj]
    lw $t4,y($t2)    # $t4=y[ii][jj]
    add $t3,$t3,$t4  # $t3=x[ii][jj]+y[ii][jj]
    sw $t3,z($t2)    # z[ii][jj]=x[ii][jj]+y[ii][jj]
  addi $t1,$t1,1  # ++jj
  b intrare_ciclu_coloane
  iesire_ciclu_coloane:
addi $t0,$t0,1  # ++ii
b intrare_ciclu_linii
iesire_ciclu_linii:
li $v0,10
syscall
##########

Exemplu: implementarea numerelor lungi; daca dorim sa lucram cu numere care
~~~~~~~~  nu incap intr-un word, putem stoca intr-un vector de word cifrele
          sale in baza 2^32, iar algoritmii de calcul cu ele vor lucra pe
          aceste reprezentari in baza 2^32;
Ilustram ideea de lucru pentru adunarea a doua numere naturale (intregi fara
 semn) de 2 word:

.data
x: .word 0xffffffff, 0xffffffff # primul nr. = 2^64-1 reprezentat pe 2 word
y: .word 0x00000001, 0xffffffff # al 2-lea nr. = 2^64-2^32+1 rep. pe 2 word
z: .word 0,0,0                  # aici vom pune suma (are max. 3 word)
 # numere sunt stocate in ordine little-endian
 # suma va fi 0x00000000, 0xffffffff, 0x00000001
 # adica numarul 0x00000001ffffffff00000000
n: .word 2       # dimensiunea vectorilor (a numerelor in word-uri)
i: .space 4      # indice
.text
main:
 li $t7,0  # carry
 li $t0,0
 sw $t0,i  # i=0 (adun de la stanga la dreapta, conform little-endian)
inceput:
 lw $t0,i
 lw $t1,n
 bge $t0,$t1,sfarsit
  sll $t0,$t0,2  # acum $t0 este offsetul in octeti al word-ului al i-lea
  lw $t1,x($t0)  # $t1=x[i]
  lw $t2,y($t0)  # $t2=y[i]
  addu $t3,$t7,$t1  # $t3=carry vechi + x[i]
  sltu $t4,$t3,$t1  # daca am carry partial atunci $t4=0, altfel $t4=1
  move $t7,$t4      #   (oricum, carry-ul partial este 0 sau 1 adica $t4)
  addu $t3,$t3,$t2  # adun la $t3 si pe y[i]
  sltu $t4,$t3,$t2  # stochez noul carry partial in $t4
  addu $t7,$t7,$t4  # acum $t7 contine carry-ul total al pozitiei i
  sw $t3,z($t0)  # z[i]=$t3=(x[i]+y[i]+carry vechi)mod 2^32
  lw $t0,i
  addiu $t0,$t0,1
  sw $t0,i       # i=i+1
  b inceput
 sfarsit:
  beqz $t7,final # daca n-am carry la ultima pozitie, am terminat
   sll $t0,$t0,2
   sw $t7,z($t0)
  final:
li $v0,10
syscall
##########
```

```
Explicatii:
- la pozitia i (=0,..,n-1) se aduna x[i]+y[i]+carry vechi si se obtine z[i]
 si un carry nou (transmis la pozitia i+1); carry-ul initial este 0, iar daca
 am un carry nenul la pozitia i=1, el se salveaza ca z[n] (altfel z[n]=0);
- se demonstreaza inductiv ca pentru orice i carry-ul generat la pozitia i
 (il notam c(i)) nu poate fi decat 0,1; notam carry-ul initial cu c(-1);
  evident, c(-1)=0;
  presupunand 0<=c(i-1)<=1, 0<=i<=n-1, si demonstram ca 0<=c(i)<=1:
    cum 0<=x[i],y[i]<=2^32-1, rezulta 0<=x[i]+y[i]+c(i-1)<=2^33-1;
    astfel, cum c(i)=(x[i]+y[i]+c(i-1)) div 2^32 (catul intreg), rezulta
      0<=c(i)<=1;
  asadar pentru orice 0<=i<=n-1 avem 0<=c(i)<=1;
  in particular z[n], care este doar c(n-1), va fi 0 sau 1;
- intrucat la pozitai i se aduna c(i-1)+x[i]+y[i], valoarea 1 a lui c(i)
 poate aparea fie in urma insumarii c(i-1)+x[i], fie in urma insumarii
 ulterioare si a lui y[i]; in program $t7 retine c(i) iar $t4 colecteaza
 cele doua carry partiale si le insumeaza in $t7;
  se poate demonstra (exercitiu) ca ambele carry partiale sunt 0 sau 1 si
 ca nu pot fi ambele 1, dar nu am exploatat asta in program;
- o dificultate care apare este cum detectam aparitia unui carry la adunarea
 a doi word (suma se face mod 2^32 si nu am carry flag ca la procesoarele
 intel care sa retina ca am avut depasire); un artificiu este compararea
 rezultatului cu unul din operanzi; mai exact:
  daca 0<=a,b<=2^32-1 sunt doua numere naturale, atunci 0<=a+b<=2^33-2;
  cazul cand suma genereaza carry este cel cand 2^32<=a+b<=2^33-1; in acest
    caz rezultatul obtinut la efectuarea instructiunii "addu" (adica
    (a+b)mod 2^32) este c=a+b-2^32; atunci va rezulta c<a,b;
  intr-adevar, daca c>=a atunci a+b-2^32>=a, adica b>=2^32 contradictie;
    la fel rezulta c<b;
  pe de alta parte, daca suma nu genereaza carry, atunci inseamna ca am avut
    0<=a+b<=2^32-1; in acest caz rezultatul obtinut la efectuarea "addu"
    (adica (a+b)mod 2^32) este c=a+b si evident avem c>=a,b;
 in concluzie: a+b genereza carry daca si numai daca "a addu b"<a (sau <b);
 pe acest principiu se bazeaza detectarea carry-urilor partiale din program;
  de exemplu secventa:

  addu $t3,$t7,$t1  # $t3=carry vechi + x[i]
  sltu $t4,$t3,$t1  # daca am carry partial atunci $t4=0, altfel $t4=1

  face "addu" intre $t7 si $t1 (adica intre c(i-1) si x[i]) si pune
  rezultatul in $t3, apoi pune rezultatul comparatiei $t3<$t1 (sub forma
  0=fals, 1=adevarat) in $t4; avand in vedere cele de mai sus, vom avea
  $t4=1 daca si numai daca avem carry partial la adunarea lui c(i-1) si x[i]
  si, mai mult, valoarea lui $t4 este chiar carry-ul partial (care nu poate
  fi decat 0 sau 1);

Mai rapid (fara a mai incarca mereu i si n din memorie ci stocand direct in
 registri offset-urile in octeti), programul anterior se poate scrie:

.data
x: .word 0xffffffff, 0xffffffff
y: .word 0x00000001, 0xffffffff
z: .word 0,0,0
 # numere sunt stocate in ordine little-endian
 # suma va fi 0x00000000, 0xffffffff, 0x00000001
 # adica numarul 0x00000001ffffffff00000000
.text
main:
 li $t7,0  # carry
 li $t0,0  # offset-ul in octeti al word-ului curent
 li $t6,8  # numarul componentelor word * 4 (offsetul maxim)
inceput:
 bge $t0,$t6,sfarsit
  lw $t1,x($t0)   # $t1=x[i]
```

```
  lw $t2,y($t0)   # $t2=y[i]
  addu $t3,$t7,$t1  # $t3=carry vechi + x[i]
  sltu $t4,$t3,$t1  # daca am carry partial atunci $t4=0, altfel $t4=1
  move $t7,$t4      #   (oricum, carry-ul partial este 0 sau 1 adica $t4)
  addu $t3,$t3,$t2  # adun la $t3 si pe y[i]
  sltu $t4,$t3,$t2  # stochez noul carry partial in $t4
  addu $t7,$t7,$t4  # acum $t7 contine carry-ul total al pozitiei i
  sw $t3,z($t0)   # z[i]=$t3=(x[i]+y[i]+carry vechi)mod 2^32
  addiu $t0,$t0,4 # i=i+4
  b inceput
 sfarsit:
  beqz $t7,final # daca n-am carry la ultima pozitie, am terminat
   sw $t7,z($t0)
  final:
li $v0,10
syscall
##########
```

Am vazut mai sus ca la adunarea numerelor naturale A+B (intregi fara semn)
 putem detecta depasirea dupa criteriul suma < B sau suma < B.
In cazul operatiilor cu numere intregi (cu semn), putem detecta depasirea
 dupa urmatoarele criterii (depasire avem cand rezultatul nu este in
 intervalul -2^31, ..., 2^31-1):

operatie (cu semn)	operand A	operand B	rezultat indicand depasire
A+B	>= 0	>= 0	< 0
A+B	< 0	< 0	>= 0
A-B	>= 0	< 0	< 0
A-B	< 0	>= 0	>= 0

Evident, aceste metode se pot folosi doar atunci cand operatia este efectuata
cu instructiuni care nu genereaza exceptii in caz de depasire (ci doar produc
un rezultat alterat).

Exemplu: inmultirea unui numar natural lung (multi word) cu un numar natural
~~~~~~~~ de un word:

```
.data
x: .word 0x20000000, 0x1e1e1e1e # ca numar x = 0x1e1e1e1e20000000
y: .word 0x00000011             # ca numar y = 0x11 (adica 3)
z: .word 0,0,0                  # aici va fi produsul (putea fi si .space 12)
 # numere sunt stocate in ordine little-endian
 # produsul va fi 0x20000000, 0x00000000, 0x00000002
 # adica numarul 0x0000000200000000020000000
.text
main:
 li $t0,0   # offset-ul word-ului curent
 li $t6,8   # numarul de octeti ai deinmultitului
 li $t7,0   # carry
 lw $t2,y   # inmultitorul
inceput:
 bge $t0,$t6,sfarsit
  lw $t1,x($t0)  # word-ul curent din deinmultit
  multu $t1,$t2  # (hi,lo) <- $t1 * $t2
  mflo $t3       # daca n-am avea carry, lo (i.e. $t3) s-ar pune in z($t0)
  mfhi $t4       # daca n-am avea carry, hi (i.e. $t4) ar fi noul carry
  addu $t3,$t3,$t7  # adunam la $t3 vechiul carry
  sw $t3,z($t0)     # $t3 da word-ul curent din z
  slt $t5,$t3,$t7   # aflam (ca la prog. precendent) carry-ul partial
  addu $t7,$t4,$t5  # noul carry = carry-ul partial + vechiul hi
                    #  (se demonstraza ca nu putem avea si aici carry)
  addu $t0,$t0,4
  b inceput
```
~~~~~~~~

```
 sfarsit:
  beqz $t7,final
  sw $t7,z($t0)
final:
li $v0,10
syscall
##########

Explicatii:
  daca n este nr. de word-uri ale deinmultitului x si 0<=i<=n-1 este word-ul
 curent, daca notam c(i) carry-ul generat la pozitia i si c(-1) carry-ul
 initial (evident 0), avem z[i] = (x[i]*y + c(i-1)) mod 2^32 si
 c(i) = (x[i]*y + c(i-1)) div 2^32; in final z[n] va primi valoarea c(n-1);
  in program noi calculam succesiv:
  $t3 = x[i]*y mod 2^32
  $t4 = x[i]*y div 2^32
  z[i]= (x[i]*y mod 2^32 + c(i-1)) mod 2^32
  $t5 = (x[i]*y mod 2^32 + c(i-1)) div 2^32
  c(i)= $t4 + $t5 = x[i]*y div 2^32 + (x[i]*y mod 2^32 + c(i-1)) div 2^32
    (iar aici am spus ca nu putem avea depasire)
 avand in vedere formulele (a+b) mod k = (a mod k + b) mod k si
  (a+b) div k = a div k + ((a mod k) + b) div k (a,b,k naturale, k>0)
  rezulta ca z[i] este corect calculat; de asemenea va rezulta ca c(i) este
  corect calculat daca demonstram ca adunarea $t4+$t5 nu avem depasire (deci
  toata suma incape in $t7); acest lucru rezulta din faptul ca avand
  0<=x[i],y<=2^32-1, avem 0<=x[i]*y<=2^64-2^33+1, deci
  0<=x[i]*y div 2^32<=2^32-2, deci 0<=$t4<=2^32-2, iar valoarea lui $t5 nu
  poate fi decat 0,1, intrucat este depasirea rezultata dintr-un "addu" (a se
  vedea explicatiile de la programul anterior), deci in total suma $t4+$t5 nu
  depaseste 2^32-1.

* Instructiuni de lucru in virgula mobila:

 In instructiunile de mai jos prin FReg, FRdest, FRsrc, FRsrc1, FRsrc2 am
notat niste registrii ai coprocesorului de virgula mobila $f0, ..., $f31; in
cazul instructiunilor in dubla precizie trebuie in plus sa fie de cod par:
$f0, $f2, ..., $f30 (si ei vor desemna o locatie de doi registri, de cod
par-impar, de exemplu $f0 va desemna perechea $f0-$f1). Vom mai nota cu cc
unul din flag-urile 0 - 7 ale coprocesorului de virgula mobila.

lwc1/ldc1   FRdest,   adr
swc1/sdc1   FRsrc,   adr
l.s/l.d   FRdest,   adr
s.s/s.d   FRsrc,   adr

## lwc1 si l.s incarca un single de la adresa "adr" in reg. FRdest;
#  ldc1 si l.d incarca un double de la adresa "adr" in reg. FRdest;
#  swc1 si s.s scrie un single din reg. FRsrc la adresa "adr";
#  sdc1 si s.d scrie un double din reg. FRsrc la adresa "adr";
## in toate cazurile transferul consta practic in copierea configuratiei
#    binare intre memorie si registri;
## in cazul double FRdest, FRsrc trebuie sa fie de cod par si se transfera
#   de fapt doua word-uri intre adresa "adr" si perechea de registri FRdest,
#   Frdest+1, respectiv FRsrc,FRsrc+1;
## "adr"poate fi: imm, et, et+/-imm, (reg), imm(reg), et(reg), et+/-imm(reg),
#   unde "imm" este o valoare imediata, "et" o eticheta, "reg" un registru
#   general (deci nu unul de virgula mobila); semnificatia expresiei "adr"
#   este ca la instructiunile lw, lb, sw, sb;
## toate sunt pseudoinstructiuni in afara de lwc1, ldc1, swc1, sdc1 in cazul
#   cand "adr" este imm(reg), caz in care toate au formatul I cu:
#                    |   op   |   rs  |FRdest |          imm           |
#                    -------------------------------------------------
#                      6 biti   5 biti  5 biti          16 biti
#   unde op este: 0x31(lwc1)/0x35(ldc1)/0x39(swc1)/0x3d(sdc1);
```

```
## practic am constatat ca transferul consta in copierea propriuzisa a
#   configuratiei de biti (fara a o interpreta ca numar) intre memorie si
#   registri;

in cele ce urmeaza, pentru simplitate, vom omite sa mai mentionam ca in cazul
double sunt afectate cate doua word-uri si sunt implicati cate doi registri,
de cod par-impar, iar in instructiune este scris doar cel de cod par;

li.s/li.d   FRdest,   imm
## incarcare valoare imediata single/double;
## efect: FRdest <- imm
## valoarea imediata trebuie scrisa in virgula mobila - de exemplu 2.0, nu 2;
## sunt pseudoinstructiuni (?);

mfc1/mtc1  rt, reg

## muta din / in coprocesorul de virgula mobila (coprocesorul 1);
#  sunt variantele pentru z = 1 ale instructiunilor mfcz/mtcz prezentate
#   in sectiunea "Instructiuni de transfer date intre memorie si registri";
## efectueaza:
#  mfc1: copiaza word-ul din reg. "reg" al coprocesorului 1
#         in reg. "rt" al CPU;
#  mtc1: copiaza word-ul din reg. "rt" al CPU
#         in reg. "reg" al coprocesorului 1;
#  practic se copiaza config. binara intre "rt" si "reg", fara conversie
#   intre formatul de intreg si cel de single;
#  registrii pot fi indicati prin $cod sau $nume, avand grija se se
#   foloseasca numele specifice registrilor din UCP sau coprocesorul 1 - de
#   ex. reg $8 al UCP este $t0 iar reg. $8 al coprocesorului de virgula
#   mobila este $f8;
## format intern:
#                     |  0x11  |  0/4  |  rt   |  reg  |       0        |
#                     -------------------------------------------------
#                       6 biti   5 biti  5 biti  5 biti      11 biti
## exemplu:
#   .text
#   main:
#   li.s $f1,0.5
#    # 0.5 se reprezinta ca single pe un word astfel: 0x3f000000
#   mfc1 $t0, $f1
#    # instr. echiv. cu: mfc1 $t0, $1
#    #  (se ia reg. 1 din coprocesorul de virgula mobila, adica $f1, nu cel
#    #  de uz general, adica $at)
#    # acum $t1 = 0x3f000000 (i.e. nr. 1056964608 reprezentat ca intreg)
#    #  (deci s-a copiat config. binara din $f1 in $t0, fara conversie de la
#    #  formatul in virgula mobila la cel de intreg)
#   li $t1,1061158912
#    # 1061158912 se reprezinta ca intreg pe un word astfel: 0x3f400000
#   mtc1 $t1, $f2
#    # instr. echiv. cu: mtc1 $t1, $2
#    #  (se ia reg. 2 din coprocesorul de virgula mobila, adica $f2, nu cel
#    #  de uz general, adica $v0)
#    # acum $f2 = 0x3f400000 (i.e. nr. 0.75 reprezentat ca single)
#    #  (deci s-a copiat config. binara din $t1 in $f2, fara conversie de la
#    #  formatul de intreg la cel in virgula mobila)
#   li $v0,10
#   syscall

mfc1.d/mtc1.d  reg,  FReg

## muta double din / in coprocesorul de virgula mobila;
## efectueaza:
#  mfc1.d: copiaza double-ul din perechea de registri (FReg, FReg+1)
#   ai coprocesorului de virgula mobila in perechea de registri (reg, reg+1)
```

```
#   ai CPU;
#  mtc1.d: copiaza double-ul din perechea de registri (reg, reg+1) ai CPU
#   in perechea de registri (FReg, FReg+1) ai coprocesorului de virgula
#   mobila;
#  practic se copiaza (fara conversie intre formatele de flotant si intreg)
#   un word intre FReg si reg si un word intre FReg+1 in reg+1;
#  de asemenea, am observat ca nu este necesar ca FReg sa fie de cod par;
#  registrii pot fi indicati prin $cod sau $nume, avand grija se se
#   foloseasca numele specifice registrilor din UCP sau coprocesorul 1 - de
#   ex. reg $8 al UCP este $t0 iar reg. $8 al coprocesorului de virgula
#   mobila este $f8;
## toate sunt pseudoinstructiuni;
## exemplu:
#   mfc1.d  $t0,  $f2
#  efectueaza: $t0 <- $f2, $t1 <- $f3
#  (copiere de config. word, fara conversii)

c.eq.s/c.eq.d   cc   FRsrc1,   FRsrc2
c.le.s/c.le.d   cc   FRsrc1,   FRsrc2
c.lt.s/c.lt.d   cc   FRsrc1,   FRsrc2

## teste cu setarea flag-ului cc;
## efectueaza: test daca valorile continute in FRsrc1, FRsrc2 sunt in relatia
#   = (eq), <= (le), < (lt),
#   si seteaza flag-ul de cod cc (0 - 7) al coprocesorului 1 la valoarea 1
#   (=adevarat)/0 (=fals); cc se poate omite si atunci se considera 0;
## are format R cu: |  0x11  | 0x10/0x11 | FRSrc2 | FRsrc1 | cc | 0 |FC|  x |
#                   ------------------------------------------------------------
#                      6b         5b         5b       5b     3b  2b  2b  4b
#  unde x este 0x2 (eq), 0xe (le), 0xc (lt),
## testele cu PCSpim au aratat ca mereu se considera cc=0 si FC=11;
## flagul setat de aceste instructiuni poate fi ulterior testat cu bcif,bc1t;

bczt/bczf   cc   eticheta
  (unde z este 0, 1, 2, 3 - practic, am constatat ca PCSpim accepta doar 1,2)
## ramificare conditionata de flagul cc al coprocesorului z;
## efect: daca flagul cc al coprocesorului z este 1 (true)/0 (false)
#         atunci salt la eticheta
## daca cc lipseste se considera 0;
## format intern:   |  0x1z  |   8   |cc|1/0 |          depl          |
#                   ---------------------------------------------------
#                      6b       5b    3b  2b            16b
#  unde depl este numarul de instructiuni masina (word-uri) peste care se
#   sare, numarand inclusiv instructiunea bczt/bczf curenta - acest numar
#   este determinat de compilator;
## foarte utile sunt formele:
#
#     bc1t/bc1f eticheta
#
## care testeaza flagul 0 al coprocesorului 1 (coprocesorul de virgula
#   mobila) iar daca este 1 (true)/0 (false) se sare la eticheta; cu
#   aceste forme ale instructiunii putem face ramificari conditionate de
#   rezultatul comparatiilor efectuate cu c.eq.s/c.eq.d/c.le.s/c.le.d/
#   c.lt.s/c.lt.d;

mov.s/mov.d   FRdest,   FRsrc

## copiere single/double intre registri de virgula mobila
## efectueaza: FRdest <- FRsrc
## are format R cu: |  0x11  | 0x10/0x11 |   0   | FRsrc | FRdest |  0x6  |
#                   ------------------------------------------------------
#                      6b         5b         5b      5b      5b       6b

movf.s/movf.d   FRdest,   FRsrc,   cc
```

```
## copiere single/double intre registri de virgula mobila conditionata de un
#    flag false;
## efectueaza: daca flag-ul cc are valoarea 0 (false) atunci FRdest <- FRsrc;
#  cc se poate omite si atunci se considera ca este vorba de flag-ul 0;
## formatul intern: |  0x11  | 0x10/0x11 | cc | 0 | FRsrc | FRdest | 0x11 |
#                   ------------------------------------------------------
#                      6b         5b       3b  2b     5b      5b      6b

movt.s/movt.d   FRdest,   FRsrc,   cc

## copiere single/double intre registri de virgula mobila conditionata de un
#    flag true;
## efectueaza: daca flag-ul cc are valoarea 1 (true) atunci FRdest <- FRsrc;
#  cc se poate omite si atunci se considera ca este vorba de flag-ul 0;
## formatul intern: |  0x11  | 0x10/0x11 | cc | 1 | FRsrc | FRdest | 0x11 |
#                   ------------------------------------------------------
#                      6b         5b       3b  2b     5b      5b      6b

ceil.w.s/ceil.w.d   FRdest,   FRsrc

## ceil single/double;
## efectueaza partea intreaga superioara (ceil) a lui FRsrc, o converteste
#   intr-o valoare in virgula fixa pe 32 biti si o pune in FRdest;
## are format R cu: |  0x11  | 0x10/0x11 |    0   | FRsrc  | FRdest |  0xe  |
#                   ------------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b
## practic am constatat ca valoarea intreaga pusa in FRdest este codificata
#   ca intreg, nu ca un numar in virgula mobila

floor.w.s/floor.w.d   FRdest,   FRsrc

## floor single/double;
## efectueaza partea intreaga inferioara (floor) a lui FRsrc, o converteste
#   intr-o valoare in virgula fixa pe 32 biti si o pune in FRdest;
## are format R cu: |  0x11  | 0x10/0x11 |    0   | FRsrc  | FRdest |  0xf  |
#                   ------------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b
## practic am constatat ca valoarea intreaga pusa in FRdest este codificata
#   ca intreg, nu ca un numar in virgula mobila

trunc.w.s/trunc.w.d   FRdest,   FRsrc

## trunchiere single/double;
## efectueaza calculul valorii trunchiate a lui FRsrc, o converteste
#   intr-o valoare in virgula fixa pe 32 biti si o pune in FRdest;
## are format R cu: |  0x11  | 0x10/0x11 |    0   | FRsrc  | FRdest |  0xd  |
#                   ------------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b
## practic am constatat ca valoarea intreaga pusa in FRdest este codificata
#   ca intreg, nu ca un numar in virgula mobila

round.w.s/round.w.d   FRdest,   FRsrc

## rotunjire single/double;
## efectueaza calculul valorii rotunjite a lui FRsrc, o converteste
#   intr-o valoare in virgula fixa pe 32 biti si o pune in FRdest;
## are format R cu: |  0x11  | 0x10/0x11 |    0   | FRsrc  | FRdest |  0xc  |
#                   ------------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b
## practic am constatat ca valoarea intreaga pusa in FRdest este codificata
#   ca intreg, nu ca un numar in virgula mobila

cvt.d.w/cvt.d.s   FRdest,   FRsrc
```

```
## conversie intreg->double/single->double;
## efectueaza: converteste intregul/single-ul (deci flotant in simpla
#   precizie) din FRsrc intr-un double (deci flotant in dubla precizie) si-l
#   pune in FRdest;
## are format R cu: |  0x11  | 0x14/0x10 |    0   | FRsrc  | FRdest |  0x21 |
#                   ---------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b

cvt.s.w/cvt.s.d   FRdest,   FRsrc

## conversie intreg->single/double->single;
## efectueaza: converteste intregul/double-ul (deci in dubla precizie) din
#   FRsrc intr-un single (deci flotant in simpla precizie) si-l pune in
#   FRdest;
## are format R cu: |  0x11  | 0x14/0x11 |    0   | FRsrc  | FRdest |  0x20 |
#                   ---------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b

cvt.w.s/cvt.w.d   FRdest,   FRsrc

## conversie double->intreg/single->intreg;
## efectueaza: converteste double-ul/single-ul din FRsrc intr-un intreg si-l
#   pune in FRdest;
## are format R cu: |  0x11  | 0x10/0x11 |    0   | FRsrc  | FRdest |  0x24 |
#                   ---------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b

Practic am constatat ca la toate instructiunile "cvt" ce fac conversie in/din
intreg, valoarea intreaga (dpv. matematic) destinatie/sursa este codificata
in registrul respectiv ca intreg, nu ca numar in virgula mobila.

abs.s/abs.d   FRdest,   FRsrc

## valoare absoluta single/double;
## efectueaza: FRdest <- |FRsrc|;
## are format R cu: |  0x11  | 0x10/0x11 |   0   | FRsrc | FRdest |  0x5  |
#                   ---------------------------------------------------------
#                      6b         5b         5b      5b      5b       6b

neg.s/neg.d   FRdest,   FRsrc

## opusul single/double;
## efectueaza: FRdest <- - FRsrc;
#  avand in vedere modul de reprezentare a numerelor in virgula mobila,
#    practic configuratia destinatie difera de cea sursa doar prin bitul cel
#    mai semnificativ (care se inlocuieste cu negatul sau);
## are format R cu: |  0x11  | 0x10/0x11 |   0   | FRsrc | FRdest |  0x7  |
#                   ---------------------------------------------------------
#                      6b         5b         5b      5b      5b       6b

add.s/add.d   FRdest,   FRsrc1,   FRsrc2

## adunare single/double;
## efectueaza: FRdest <- FRsrc1 + FRsrc2;
## are format R cu: |  0x11  | 0x10/0x11 | FRsrc2 | FRsrc1 | FRdest |   0   |
#                   ---------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b

sub.s/sub.d   FRdest,   FRsrc1,   FRsrc2

## scadere single/double;
## efectueaza: FRdest <- FRsrc1 - FRsrc2;
## are format R cu: |  0x11  | 0x10/0x11 | FRsrc2 | FRsrc1 | FRdest |  0x1  |
```

```
#                   ------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b

mul.s/mul.d   FRdest,   FRsrc1,   FRsrc2

## inmultire intre doua single/double
## efectueaza: FRdest <- Frsrc1 * Frsrc2
## are format R cu: |  0x11  | 0x10/0x11 | FRsrc2 | FRsrc1 | FRdest |  0x2  |
#                   ------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b

div.s/div.d   FRdest,   FRsrc1,   FRsrc2

## impartire intre doua single/double
## efectueaza: pune in FRdest catul exact (ca single/double) impartirii lui
#   FRsrc1 la FRsrc2
## are format R cu: |  0x11  | 0x10/0x11 | FRsrc2 | FRsrc1 | FRdest |  0x3  |
#                   ------------------------------------------------------
#                      6b         5b         5b       5b       5b       6b

sqrt.s/sqrt.d   FRdest,   FRsrc

## radacina patrata single/double;
## efectueaza: FRdest <- radacina patrata a lui FRsrc;
## are format R cu: |  0x11  | 0x10/0x11 |   0   | FRsrc | FRdest |  0x4  |
#                   ------------------------------------------------------
#                      6b         5b         5b      5b      5b       6b

Exemplu: aratam ca instructiunile de transfer intre registrii de virgula
~~~~~~~~  mobila si memorie copiaza configuratia de biti fara conversie,
          la fel ca instructiunile de transfer intre registrii generali si
          memorie (la rulare se va urmari zona de date statice).
.data
s1: .word 0x40980000  # poate fi intregul 1083703296 sau single-ul 4.75
i1: .space 4
r1: .space 4
i2: .space 4
r2: .space 4
s2: .word 0, 0x40980000 # tinand cont de little-endian, cei 2 word
rr1: .space 8           #  (adica 0x4098000000000000) pot insemna
rr2: .space 4           #  intregul 4654470214887407616 sau
rr3: .space 8           #  flotantul double 1536
.text
main:
lw $t0,s1    # incarca (copiaza) config. binara 0x40980000 in $t0
lwc1 $f0,s1  # incarca (copiaza) config. binara 0x40980000 in $f0
sw $t0,i1    # scrie (copiaza) config. binara 0x40980000 din $t0 la adresa i1
swc1 $f0,r1  # scrie (copiaza) config. binara 0x40980000 din $f0 la adresa r1
li $t1,1
li.s $f1,1.0
add $t0,$t0,$t1
add.s $f0,$f0,$f1
  # pt. reg. de virgula mobila nu pot folosi "li" ci "li.s" sau "li.d" iar
  #  1 trebuie scris ca flotant: 1.0
sw $t0,i2    # scrie config. binara din $t0 la adresa i2
swc1 $f0,r2  # scrie config. binara din $f0 la adresa r2
# indiferent daca instructiunile de transfer sunt pentru reg. generali sau
#  cei de virgula mobila, ele copiaza din/in memorie in/din registri config.
#  de biti, fara a o interpreta ca numar; astfel rezulta i1: 0x40980000,
#  r1: 0x40980000
# operatia de adunare insa interpreteaza aceeasi config. 0x40980000 in mod
#  diferit: intregul 1083703296 ("add"), flotantul 4.75 ("add.s"); de aceea
#  in $t0 si mai apoi la adr. i2 (prin copierea bit cu bit) obtinem config.
#  0x40980001 (intregul 1083703297) iar in $f0 si mai apoi la adr. r2 (prin
```

```
#  copierea bit cu bit) obtinem config. 0x40b80000 (flotantul single 5.75)
#####
ldc1 $f0,s2  # copiaza 2 word de la adresa s2 in perechea $f0,$f1
sdc1 $f0,rr1 # copiaza 2 word din perechea $f0,$f1 la adresa rr1
  # acum la adr. rr1 avem: 0, 0x40980000
swc1 $f1,rr2 # copiaza 1 word din $f1 la adr. rr2; deci rr2: 0x40980000
li.d $f2,1.0 # incarca nr. 1 ca double in perechea de registri $f2,$f3
  # nu puteam scrie "li.d $f1,1.0" deoarece valorile double se stocheaza
  # doar in perechi de reg. de virgula mobila de cod par-impar, nu impar-par
add.d $f0,$f0,$f2
sdc1 $f0,rr3 # copiaza 2 word din perechea $f0,$f1 la adresa rr2
  # acum rr3: 0x00000000  0x40980400 adica 1537 ca double (nu ca intreg)
li $v0,10
syscall
##########

Exemplu: aratam ca instructiunile "ceil", "floor" si "cvt" ce fac conversie
~~~~~~~~  in/din intreg, valoarea intreaga (dpv. matematic) destinatie/sursa
    este codificata in registrul respectiv ca intreg, nu ca numar in virgula
    mobila (la rulare se va urmari continutul zonei de date statice).

.data
r1: .space 4
r2: .space 4
r3: .space 4
r4: .space 4
r5: .space 4
r6: .space 4
r7: .space 4
r8: .space 4
r9: .space 4
r10: .space 4
.text
main:
li.s $f0,5.0
swc1 $f0,r1
li.s $f0,4.75
ceil.w.s $f1,$f0
swc1 $f1,r2
floor.w.s $f2,$f0
swc1 $f2,r3
# desi la adresele r1, r2 e scrisa aceeasi valoare matematica 5, in primul
#  caz este stocata in virgula mobila, iar in al doilea caz ca intreg:
#  r1: 0x40a00000, r2: 0x00000005; la adresa r3 este scrisa valoarea
#  matematica 4 stocata ca intreg: 0x00000004;
# deci ceil.w.s, floor.w.s produc un intreg reprezentat ca intreg, iar swc1
#  copiaza configuratia de biti ca atare - astfel, din $f0 este copiata
#  configuratia 0x40a00000 care este 5 ca flotant, din $f1 configuratia
#  0x00000005 care este 5 ca intreg, iar din $f2 configuratia 0x00000004
#  care este 4 ca intreg
##########
li.s $f0,4.75    # incarca in $f0 4.75 reprezentat ca single
cvt.w.s $f1,$f0  # pune in $f1 valoarea 4 reprezentat ca intreg
swc1 $f1,r4      # la adresa r4 ajunge 4 reprezentat ca intreg (pe 1 word)
cvt.s.w $f0,$f1  # pune in $f0 valoarea 4 reprezentata ca single
swc1 $f0,r5      # la adresa r5 ajunge 4 reprezentat ca single
# rularea confirma ca la conversia in/din intreg valoarea intreaga din punct
#  de vedere matematic destinatie/sursa este codificata ca intreg, nu ca
#  flotant; astfel "cvt.w.s" plecand de la single-ul 4.75 produce intregul
#  (nu single-ul) 4 iar acesta ajunge la adresa r4 (reamintim ca "swc1"
#  copiaza configuratia de biti ca atare din registru in memorie, fara
#  conversie); de asemenea, "cvt.s.w" plecand de la intregul 4 (stocat inca
#  in $f1) produce single-ul 4 iar acesta ajunge la adresa r5;
#in final r4: 0x00000004  r5: 0x40800000
```

```
##########
li.s $f0,-5.0     # incarca in $f0 -5 reprezentat ca single
swc1 $f0,r6       # scrie la adresa r6 pe -5 reprezentat ca single
li.s $f0,-4.75    # incarca in $f0 -4.75 reprezentat ca single
ceil.w.s $f1,$f0  # pune in $f1 -4 reprezentat ca intreg
swc1 $f1,r7       # scrie la adresa r7 -4 reprezentat ca intreg
floor.w.s $f2,$f0 # pune in $f2 -5 reprezentat ca intreg
swc1 $f2,r8       # scrie la adresa r8 -5 reprezentat ca intreg
li.s $f0,-4.75   # incarca in $f0 -4.75 reprezentat ca single
cvt.w.s $f1,$f0  # pune in $f1 valoarea -4 reprezentat ca intreg
swc1 $f1,r9      # scrie la adresa r9 -4 reprezentat ca intreg (pe 1 word)
cvt.s.w $f0,$f1  # pune in $f0 valoarea -4 reprezentata ca single
swc1 $f0,r10     # scrie la adresa r10 -4 reprezentat ca single
# repetarea celor de mai sus cu numere negative; se confirma aceeasi
#  regula: valoarea intreaga destinatie/sursa la instructiunile de conversie
#  in/din intreg si la ceil, floor este reprez. ca intreg, nu ca flotant;
# in final r6: 0xc0a00000  r7: 0xfffffffc  r8: 0xfffffffb
#          r9: 0xfffffffc  r10: 0xc0800000
li $v0,10
syscall
##########
```

Exemplu: aratam ca conversiile in intreg se fac cu eliminarea zecimalelor
~~~~~~~~ (la rulare se va urmari continutul zonei de date statice).

```
.data
r1: .space 4
r2: .space 4
r3: .space 4
r4: .space 4
r5: .space 4
r6: .space 4
.text
main:
li.s $f1,2.3
li.s $f2,2.5
li.s $f3,2.6
cvt.w.s $f0,$f1
swc1 $f0,r1      # la adresa r1 se scrie 2 reprezentat ca intreg: 0x00000002
cvt.w.s $f0,$f2
swc1 $f0,r2      # la adresa r2 se scrie 2 reprezentat ca intreg: 0x00000002
cvt.w.s $f0,$f3
swc1 $f0,r3      # la adresa r3 se scrie 2 reprezentat ca intreg: 0x00000002
li.s $f1,-2.3
li.s $f2,-2.5
li.s $f3,-2.6
cvt.w.s $f0,$f1
swc1 $f0,r4      # la adresa r4 se scrie -2 reprezentat ca intreg: 0xfffffffe
cvt.w.s $f0,$f2
swc1 $f0,r5      # la adresa r5 se scrie -2 reprezentat ca intreg: 0xfffffffe
cvt.w.s $f0,$f3
swc1 $f0,r6      # la adresa r6 se scrie -2 reprezentat ca intreg: 0xfffffffe
li $v0,10
syscall
##########
```

Exemplu: comparatie intre floor/ceil si round/trunc (se ruleaza pas cu pas,
~~~~~~~~   urmarind segmentul de date statice):

```
.data
r1: .space 4
r2: .space 4
r3: .space 4
r4: .space 4
```

```
.text
main:
li.s $f0,2.3
floor.w.s $f1, $f0
swc1 $f1, r1       # r1: 0x00000002 (2 ca intreg)
ceil.w.s $f1, $f0
swc1 $f1, r2       # r2: 0x00000003 (3 ca intreg)
round.w.s $f1, $f0
swc1 $f1, r3       # r3: 0x00000002 (2 ca intreg)
trunc.w.s $f1, $f0
swc1 $f1, r4       # r4: 0x00000002 (2 ca intreg)
#####
li.s $f0,2.5
floor.w.s $f1, $f0
swc1 $f1, r1       # r1: 0x00000002
ceil.w.s $f1, $f0
swc1 $f1, r2       # r2: 0x00000003
round.w.s $f1, $f0
swc1 $f1, r3       # r3: 0x00000003
trunc.w.s $f1, $f0
swc1 $f1, r4       # r4: 0x00000002
#####
li.s $f0,2.7
floor.w.s $f1, $f0
swc1 $f1, r1       # r1: 0x00000002
ceil.w.s $f1, $f0
swc1 $f1, r2       # r2: 0x00000003
round.w.s $f1, $f0
swc1 $f1, r3       # r3: 0x00000003
trunc.w.s $f1, $f0
swc1 $f1, r4       # r4: 0x00000002
#####
li.s $f0,-2.3
floor.w.s $f1, $f0
swc1 $f1, r1       # r1: 0xfffffffd (-3 ca intreg)
ceil.w.s $f1, $f0
swc1 $f1, r2       # r2: 0xfffffffe (-2 ca intreg)
round.w.s $f1, $f0
swc1 $f1, r3       # r3: 0xffffffff (-1 ca intreg)
trunc.w.s $f1, $f0
swc1 $f1, r4       # r4: 0xfffffffe (-2 ca intreg)
#####
li.s $f0,-2.5
floor.w.s $f1, $f0
swc1 $f1, r1       # r1: 0xfffffffd
ceil.w.s $f1, $f0
swc1 $f1, r2       # r2: 0xfffffffe
round.w.s $f1, $f0
swc1 $f1, r3       # r3: 0xfffffffe
trunc.w.s $f1, $f0
swc1 $f1, r4       # r4: 0xfffffffe
#####
li.s $f0,-2.7
floor.w.s $f1, $f0
swc1 $f1, r1       # r1: 0xfffffffd
ceil.w.s $f1, $f0
swc1 $f1, r2       # r2: 0xfffffffe
round.w.s $f1, $f0
swc1 $f1, r3       # r3: 0xfffffffe
trunc.w.s $f1, $f0
swc1 $f1, r4       # r4: 0xfffffffe
#####
li $v0,10
syscall
```

```
##########

Exemplu: rezolvarea (neinteractiva a) ecuatiei de grad <= 1:
~~~~~~~~

.data
coefa: .float 2.0
coefb: .float 9.5
x: .space 4  # va contine solutia
e: .space 4  # va contine: 1 (sol. unica), 2 (inf. de sol), 3 (fara sol.)
.text
main:
li.s $f0,0.0     # $f0 <- 0.0
l.s $f1,coefa    # $f1 <- 2.0 (coefa)
l.s $f2,coefb    # $f2 <- 9.5 (coefb)
c.eq.s $f1, $f0  # daca $f1=0 setam flag-ul 0 la valoarea 1
bc1t et1         # daca flag-ul 0 are val. 1, salt la et. "et1"
 # cazul coefa nenul
neg.s $f2,$f2      # $f2 <- - coefb
div.s $f2,$f2,$f1  # $f2 <- - coefb/coefa
s.s $f2,x          # scriem sol. la adr. x
li $t0,1
sw $t0,e           # solutie unica
j sfarsit
 # cazul coefa nul
et1:
c.eq.s $f2, $f0  # daca $f2=0 setam flag-ul 0 la valoarea 1
bc1t et2         # daca flag-ul 0 are val. 1, salt la et. "et2"
 # cazul coefa nul, coefb nenul
li $t0,3
sw $t0,e           # fara solutii
j sfarsit
 # cazul coefa nul, coefb nul
et2:
li $t0,2
sw $t0,e           # infinitate de solutii
sfarsit:
li $v0,10
syscall
##########
# in final x: 0xc0980000 (-4.75 float), e: 0x00000001 (sol. unica)
##########

* Alte instructiuni:

rfe

## intoarcere din exceptie;
## efectueaza: reface regsitrul de stare;
## format intern: 0x42000010

syscall

## apel sistem;
## efectueaza: apeleaza rutina sistemului de operare;
#  cu ajutorul ei se pot accesa resurse aflate in gestiunea sistemului de
#   operare, de ex. consola;
#  inainte de apel se incarca diversi parametri in anumiti registri; de
#   exemplu in $v0 se va incarca numarul apelului sistem dorit;
#  detalii in sectiunea E ("Apeluri sistem");
## format intern: 0x0000000c

break cod
```

```
## intrerupere;
## efectueaza: apeleaza codul de exceptie (din sistemul de operare) cu
#   numarul "cod"; exceptia 1 este rezervata depanatorului;
## format intern:
#                   |    0   |           cod             |    0   |  0xd  |
#                   ---------------------------------------------------------
#                     6 biti            15 biti             5 biti  6 biti

nop

## nici o operatie;
## efectueaza: nimic;
## format intern: 0x00000000
```

D. Utilizarea stivei:
=====================

Daca o anumita prelucrare este prea complexa, putem ajunge in situatia de a nu avea suficienti registri disponibili pentru a stoca rezultatele intermediare si atunci suntem nevoiti sa le stocam temporar in memorie (cu pretul scaderii vitezei de executie a programului); in acest scop am putea folosi zone alocate static (la .data) dar e inestetic si oricum e greu de anticipat cantitatea de memorie necesara, mai ales ca poate diferi de la o executie la alta. De aceea, pentru stocarea valorilor temporare in memorie se foloseste stiva. Reamintim ca stiva creste spre adrese mici iar registrul $sp are drept rol sa retina in permanenta adresa varfului stivei.
Astfel, putem incarca (push) un word din $t0 in stiva cu secventa:

```
subu $sp,4
sw $t0,0($sp)
```

si putem descarca (pop) word-ul din varful stivei in $t0 cu secventa:

```
lw $t0,0($sp)
addu $sp,4
```

Putem incarca/descarca mai multe date in/din stiva, dar trebuie sa scadem/crestem corespunzator $sp si, evident, sa descarcam datele in ordinea inversa in care le-am incarcat (altfel nu vom recupera valorile corecte). Notam ca $sp (ca orige registru general) poate fi folosit/modificat oricum de utilizator, dar daca il folosim in alte scopuri vom pierde controlul asupra stivei.

Exemplu: Evaluarea unei expresii;
~~~~~~~~ vrem sa evaluam expresia ((x+y)*(x-y)*(x+z))%(y+z)
Varianta cu registri (se va rula pas cu pas urmarind registrii $v0, $v1, $t0, $t1, $t2):

```
.data
x: .word 2
y: .word 1
z: .word 3
.text
main:
##### calculam x+y
lw $v0,x
lw $v1,y
add $v0,$v0,$v1
move $t0,$v0   # $t0=x+y=3
##### calculam x-y
lw $v0,x
lw $v1,y
sub $v0,$v0,$v1
move $t1,$v0   # $t1=x+y=1
```
~~~~~~~~

```
##### calculam x+z
lw $v0,x
lw $v1,z
add $v0,$v0,$v1
move $t2,$v0   # $t2=x+z=5
##### calculam (x+y)*(x-y)*(x+z)
move $v0,$t0
mulo $v0,$v0,$t1
mulo $v0,$v0,$t2
move $t0,$v0   # $t0=(x+y)*(x-y)*(x+z)=15
##### calculam y+z
lw $v0,y
lw $v1,z
add $v0,$v0,$v1
move $t1,$v0   # $t1=y+z=4
##### calculam ((x+y)*(x-y)*(x+z))%(y+z)
div $t0,$t1
mfhi $v0  # $v0=((x+y)*(x-y)*(x+z))%(y+z)=3
li $v0,10
syscall
##########
```

Obs: vedem ca am avut nevoie de cate un registru pentru fiecare din cele 3 subexpresii ale expresiei ale parantezei ((x+y)*(x-y)*(x+z)); daca paranteza ar fi avut mai mutlte subexpresii (de exemplu 33 expresii) nu am fi avut suficienti registri disponibili si ar fi trebuit sa stocam rezultatele partiale si in memorie; mai mult, daca numarul subexpresiilor parantezei ar fi fost variabil (de exemplu daca am evalua ceva de forma "produs pentru i de la 0 la n din (x[i]+y)", cu n citit interactiv) necesarul de memorie suplimentara nu poate fi anticipat la scrierea programului.
Varianta care rezolva toate problemele este cea in care stocam rezultatele partiale in stiva (pentru eficientizare putem stoca unele rezultate partiale si in registri - de exemplu valorile unor subexpresii care se repeta):

```
.data
x: .word 2
y: .word 1
z: .word 3
.text
main:
########## calculam y+z si introducem in stiva
lw $t0,y
lw $t1,z
add $t0,$t0,$t1 # $t0 = y+z = 4
subu $sp,4
sw $t0,0($sp)   # acum stiva contine: $sp:4
########## calculam ((x+y)*(x-y)*(x+z)) si introducem in stiva
##### calculam x+z si introducem in stiva
lw $t0,x
lw $t1,z
add $t0,$t0,$t1 # $t0 = x+z = 5
subu $sp,4
sw $t0,0($sp)   # acum stiva contine: $sp:5,4
##### calculam x-y si introducem in stiva
lw $t0,x
lw $t1,y
sub $t0,$t0,$t1 # $t0 = x-y = 1
subu $sp,4
sw $t0,0($sp)   # acum stiva contine: $sp:1,5,4
##### calculam x+y  si introducem in stiva
lw $t0,x
lw $t1,y
add $t0,$t0,$t1 # $t0 = x+y = 3
subu $sp,4
```

```
sw $t0,0($sp)   # acum stiva contine: $sp:3,1,5,4
##### acum stiva contine valorile subexpresiilor primei expresii
##### ramane sa inmultim aceste valori
lw $t0,0($sp)
lw $t1,4($sp)
addu $sp,8
mulo $t0,$t0,$t1
subu $sp,4
sw $t0,0($sp)   # acum stiva contine: $sp:3,5,4
#####
lw $t0,0($sp)
lw $t1,4($sp)
addu $sp,8
mulo $t0,$t0,$t1
subu $sp,4
sw $t0,0($sp)   # acum stiva contine: $sp:15,4
########## acum stiva contine valorile subexpresiilor expresiei mari
########## ramane sa impartim aceste valori
lw $t0,0($sp)
lw $t1,4($sp)
addu $sp,8
div $t0,$t1
mfhi $t0
subu $sp,4
sw $t0,0($sp)   # acum stiva contine: $sp:3
#################### recuperam din stiva in $v0 intregii expresii
lw $v0,0($sp)
addu $sp,4      # $v0=3
li $v0, 10
syscall
##########
(la rularea pas cu pas se vor urmari $v0, $t0 si continutul stivei)
```

Regula de evaluare a expresiilor cu stiva este (presupunem ca operanzii si rezultatele sunt word):

- scriem expresia e sub forma e1 o1 ... on en, unde e1, ..., e2 sunt subexpresii in paranteze iar o1, ..., o2 operatori cu aceeasi prioritate si mod de asociere;
- daca o1, ..., on sunt cu asociere de la stanga la dreapta, efectuam:

```
<evaluare en si push-area rezultatului in stiva>
 ...
<evaluare e1 si push-area rezultatului in stiva>

lw $t0,0($sp)
lw $t1,4($sp)
addu $sp,8
<$t0 <- $t0 o1 $t1>
subu $sp,4
sw $t0,0($sp)
 ...
lw $t0,0($sp)
lw $t1,4($sp)
addu $sp,8
<$t0 <- $t0 on $t1>
subu $sp,4
sw $t0,0($sp)

lw $v0,0($sp)
addu $sp,4
```

- daca o1, ..., on sunt cu asociere de la dreapta la stanga, efectuam:

```
<evaluare e1 si push-area rezultatului in stiva>
```

```
  ...
  <evaluare en si push-area rezultatului in stiva>

  lw $t0,0($sp)
  lw $t1,4($sp)
  addu $sp,8
  <$t0 <- $t0 on $t1>
  subu $sp,4
  sw $t0,0($sp)
   ...
  lw $t0,0($sp)
  lw $t1,4($sp)
  addu $sp,8
  <$t0 <- $t0 o1 $t1>
  subu $sp,4
  sw $t0,0($sp)

  lw $v0,0($sp)
  addu $sp,4
```

- pentru <evaluare ei si push-area rezultatului in stiva> se procedeaza ca pentru e.

Daca operanzii/rezultatele sunt date multi-word, la fiecare etapa se va face push/pop pentru mai multe word-uri.

Regula poate fi simplificata observand ca la spargerea unei expresii in subexpresii valoarea ultimei subexpresii evaluate nu mai trebuie push-ata, deoarece este prima pop-ata. De asemenea, daca valoarea finala trebuie recuperata in $v0, in loc de registrii $t0, $t1 putem folosi $v0, $t0.
Cu aceaste modificari, programul devine (la rularea pas cu pas se vor urmari $v0, $t0 si continutul stivei):

```
.data
x: .word 2
y: .word 1
z: .word 3
.text
main:
########## calculam y+z si introducem in stiva
lw $v0,y
lw $t0,z
add $v0,$v0,$t0 # $v0 = y+z = 4
subu $sp,4
sw $v0,0($sp)   # acum stiva contine: $sp:4
########## calculam ((x+y)*(x-y)*(x+z)) (si nu mai introducem in stiva)
##### calculam x+z si introducem in stiva
lw $v0,x
lw $t0,z
add $v0,$v0,$t0 # $v0 = x+z = 5
subu $sp,4
sw $v0,0($sp)   # acum stiva contine: $sp:5,4
##### calculam x-y si introducem in stiva
lw $v0,x
lw $t0,y
sub $v0,$v0,$t0 # $v0 = x-y = 1
subu $sp,4
sw $v0,0($sp)   # acum stiva contine: $sp:1,5,4
##### calculam x+y (si nu mai introducem in stiva)
lw $v0,x
lw $t0,y
add $v0,$v0,$t0 # $v0 = x+y = 3
##### acum $v0 si stiva contin valorile subexpresiilor primei expresii
##### ramane sa inmultim aceste valori
```

```
lw $t0,0($sp)
addu $sp,4
mulo $v0,$v0,$t0  # acum $v0=3*1=3 iar stiva ($sp):5,4
#####
lw $t0,0($sp)
addu $sp,4
mulo $v0,$v0,$t0  # acum $v0=3*1*5=15 iar stiva ($sp):4
########## acum $v0 si stiva contin valorile subexpresiilor expresiei mari
########## ramane sa impartim aceste valori
lw $t0,0($sp)
addu $sp,4
div $v0,$t0
mfhi $v0
############## acum $v0 contine valoare intregii expresii 15 iar stiva e vida
li $v0, 10
syscall
##########
```

Un alt caz in care putem ajunge sa nu avem suficienti registri disponibili este parcurgerea masivelor cu multe dimensiuni - am vazut intr-un exemplu anterior (adunarea matricilor) ca pentru fiecare dimensiune avem nevoie de un registru separat, sau de o locatie statica separata. Putem reduce numarul registrilor necesari salvandu-le valorile in stiva si refolosindu-i.

Exemplu: suma elementelor unui masiv 3-dimensional, pe care il parcurgem cu
~~~~~~~~ un singur indice.
Translatam urmatorul program (pseudo) C:

```
int x[2][3][4]=
  {{{0x1,0x2,0x3,0x4},{0x5,0x6,0x7,0x8},{0x9,0xa,0xb,0xc}},
  {{0xd,0xe,0xf,0x10},{0x11,0x12,0x13,0x14},{0x15,0x16,0x17,0x18}}},
  n1=2, n2=3, n3=4, s=0;
for(i=0;i<n1;++i){
  push i,n1;
  for(i=0;i<n2;++i){
    push i,n2;
    for(i=0;i<n3;++i) s += x[stiva[12]][stiva[4]][i];
    pop n2,i;
  }
  pop n1,i;
}

.data
x: .word 0x1,0x2,0x3,0x4     # cele 24 elemente sunt alocate in memorie
   .word 0x5,0x6,0x7,0x8     #  succesiv, dar numai adresa primului este
   .word 0x9,0xa,0xb,0xc     #  asociata unei etichete, anume x
   .word 0xd,0xe,0xf,0x10
   .word 0x11,0x12,0x13,0x14
   .word 0x15,0x16,0x17,0x18
n1:.word 2
n2:.word 3
n3:.word 4
s: .space 4
.text
main:
lw $t1,n1
li $t0,0
intrare1:
bge $t0,$t1,iesire1
  subu $sp,8
  sw $t0,0($sp)
  sw $t1,4($sp) # push i1,n1; stiva este ($sp):i1,n1
    lw $t1,n2
    li $t0,0
```
~~~~~~~~

```
    intrare2:
    bge $t0,$t1,iesire2
      subu $sp,8
      sw $t0,0($sp)
      sw $t1,4($sp) # push i2,n2; stiva este ($sp):i2,n2,i1,n1
        lw $t1,n3
        li $t0,0
        intrare3:
        bge $t0,$t1,iesire3
        # indicele in vectorul prin care este liniarizat masivul
        #  corespunzator pozitiei [i1][i2][i3] in masiv este
        #  i*n2*n3+j*n3+k = ((i1*n2)+i2)*n3+i3
        # calculam acest indice in $v0 tinand cont ca
        #  $t0=i3, $t1=n3 iar stiva este ($sp):i2,n2,i1,n1
          lw $v0,8($sp)     # $v0=i1
          lw $t2,4($sp)     # $t2=n2
          mulou $v0,$v0,$t2 # $v0=i1*n2
          lw $t2,0($sp)     # $t2=i2
          addu $v0,$v0,$t2  # $v0=(i1*n2)+i2
          mulou $v0,$v0,$t1 # $v0=((i1*n2)+i2)*n3
          addu $v0,$v0,$t0  # $v0=((i1*n2)+i2)*n3+i3
          sll $v0,$v0,2     # acum $v0 este distanta in octeti a elementului
          lw $v0,x($v0)     # $v0=x[i1][i2][i3]
          lw $t2,s          # $t2=suma partiala
          addu $v0,$v0,$t2
          sw $v0,s          # s+=x[i1][i2][i3]
          addiu $t0,$t0,1  # ++i
          b intrare3
        iesire3:
      lw $t0,0($sp)
      lw $t1,4($sp)
      addu $sp,8     # pop n2,i2; stiva este ($sp):i1,n1
      addiu $t0,$t0,1  # ++i
      b intrare2
    iesire2:
  lw $t0,0($sp)
  lw $t1,4($sp)
  addu $sp,8     # pop n1,i; stiva este vida
  addiu $t0,$t0,1  # ++i
  b intrare1
iesire1:
li $v0,10
syscall
##########
```

E. Apeluri sistem:
==========

Pentru a ajunge la o resursa aflata in gestiunea sistemului de operare programul trebuie sa treaca prin acesta - el nu poate accesa resursa direct ci trebuie sa solicite sistemului de operare sa acceseze resursa pentru el. In acest scop programul apeleaza o rutina din sistemul de operare (deci neinclusa la compilare in program), numita intrerupere sofware sau apel sistem, transmitandu-i prin intermediul parametrilor ce doreste sa faca cu resursa, iar rutina respectiva va accesa resursa, va executa sarcina dorita si (eventual) va returna programului un raspuns.

In masina virtuala PCSpim, pe langa programul nostru se incarca automat si un nucleu de sistem de operare (kernel). Apelurile sistem ale acestuia se apeleaza cu "syscall" (instructiune codata intern 0x0000000c), dupa ce in prealabil am incarcat parametrii actuali corespunzatori (de regula in registri).

O resursa a masinii virtuale care se poate accesa via syscall este consola virtuala. Deci, daca vrem ca programul nostru sa citeasca/afiseze interactiv date de la/la consola trebuie sa folosim syscall.

Variantele de apelare ale lui syscall si functia (efectul) lor sunt:

print int:
parametri: $v0=1, $a0=integer (deci intreg cu semn)
efect: afisaza intregul din $a0

print float:
parametri: $v0=2, $f12=single (deci flotant in simpla precizie)
efect: afisaza nr. single respectiv

print double:
parametri: $v0=3, $f12=double (deci flotant in dubla precizie)
(de fapt double-ul este stocat in perechea $f12,$f13)
efect: afisaza nr. double respectiv

print string:
parametri: $v0=4, $a0=adresa unui string terminat cu caracterul nul
efect: afisaza stringul (pana la intalnirea caracterului nul)

read int:
parametri: $v0=5
efect: citeste de la consola un integer (deci intrag cu semn) si-l returneaza
tot in $v0

read float:
parametri: $v0=6
efect: citeste de la consola un single (deci flotant in simpla precizie) si-l
returneaza in $f0

read double:
parametri: $v0=7
efect: citeste de la consola un double (deci flotant in dubla precizie) si-l
returneaza in $f0 (de fapt in perechea $f0,$f1)

Atentie: in cazul read int, read float, read double se citeste de fapt o
linie intreaga de la consola (pana la newline inclusiv), se preia doar
primul numar din ea, iar restul liniei se ignora.

read string:
parametri: $v0=8, $a0=adresa zonei destinatie, $a1=lung. maxima a str. citit
efect: citeste de la consola un sir de cel mult $a1-1 caractere si-l pune in
memorie incepand de la adresa $a0 adaugand un caracter nul; daca in
linia introdusa de la consola sunt mai putin de $a1-1 caractere se
citesc doar aceste caractere (inclusiv newline) si apoi se adauga
caracterul nul; efectul este asemanator functiei "fgets" din C;

Atentie: programele care folosesc aceste apeluri de sistem pentru a citi de
la consola nu trebuie sa utilizeze adresarea pentru I/O transpusa in
memorie. In acest scop, inainte de a le rula, setam masina PCSpim a.i.
in meniul Simulator/Settings sa NU fie bifat Mapped I/O

sbrk:
parametri: $v0=9, $a0=cantitate
efect: aloca (dinamic) un bloc de $a0 octeti si returneaza adresa lui in $v0;
astfel sistemul adauga (dinamic) memorie suplimentara spatiului de
adresare al programului;

exit:
parametri: $v0=10
efect: opreste programul din executie

Exemplu:
~~~~~~~~
~~~~~~~~

```
.data
i1: .word 0x00000002  # adica 2 ca nr. intreg
i2: .word 0xfffffffe  # adica -2 ca nr. intreg
r1: .word 0x3f000000  # adica 0.5 ca nr. single
r2: .word 0x00000000 0xbfe00000 # adica -0.5 ca nr. double (de 2 word)
s1: .asciiz "abc"
s2: .byte 'x','y','z',0
i3: .space 4
i4: .space 4
r3: .space 4
r4: .space 8
s3: .word 0xffffffff, 0xffffffff
s4: .word 0xffffffff, 0xffffffff
.text
main:
# afisam intregul cu semn indicat de i1 (adica 2)
li $v0,1
lw $a0,i1
syscall
# afisam intregul cu semn indicat de i2 (adica -2)
li $v0,1
lw $a0,i2
syscall
# afisam single-ul indicat de r1 (adica 0.50000000)
li $v0,2
lwc1 $f12,r1
syscall
# afisam double-ul indicat de r2 (adica -0.50000000)
li $v0,3
l.d $f12,r2
syscall
# afisam stringul indicat de s1 (adica "abc")
li $v0,4
la $a0,s1
syscall
# afisam stringul indicat de s21 (adica "xyz" - vedem ca se term. cu nul)
li $v0,4
la $a0,s2
syscall
# citim un intreg
li $v0,5
syscall
sw $v0,i3 # daca tastam 2, vom gasi la adresa i3 word-ul 0x00000002
# citim un alt intreg
li $v0,5
syscall
sw $v0,i4 # daca tastam -2, vom gasi la adresa i4 word-ul 0xfffffffe
# citim un single
li $v0,6
syscall
swc1 $f0,r3 # daca tastam 0.5, vom gasi la adresa r3 word-ul 0x3f000000
            # insemnand ca single 0.5
# citim un double
li $v0,7
syscall
s.d $f0,r4 # daca tastam -0.5, vom gasi la adresa r4 word-urile
            # 0x00000000 0xbfe00000 insemnand ca double -0.5
            # (reprezentarea lui -0.5 ca double (64 biti) este de fapt
            # 0xbfe0000000000000, dar in little-endian word-ul hi e ultimul)
# citim un string
li $v0,8
la $a0,s3
li $a1,4
```

```
syscall # daca tastam abcde<ENTER>, vom gasi la adresa s3: 'a','b','c',0
        #  adica wordul 0x00636261 (in loc de 0xffffffff)
# citimd un alt string
li $v0,8
la $a0,s4
li $a1,4
syscall # daca tastam ab<ENTER>, vom gasi la adresa s4: 'a','b','\n',0
        #  adica wordul 0x000a6261 (in loc de 0xffffffff)
# terminam programul
li $v0,10
syscall
##########
```

(la rulare se va urmari consola si continutul care initial era .space;
 notam ca pe diverse simulatoare MIPS programul functioneaza diferit, chiar
 defectuos)

Observatie: la print int, print float, print double, print string nu se emite
 si un newline dupa valoarea afisata, asa ca prima parte a programului
 afisaza:

 2-2abcxyz...

 Astfel, daca vrem ca afisarea sa treaca la linie noua trebuie sa emitem
 explicit un newline, de exemplu sa afisam un string ce contine octetii 10,0.

De exemplu programul:

```
.data
nl: .byte 10,0   # sau nl: .asciiz "\n"
x: .word 1,2,3
.text
main:
li $v0,1
lw $a0,x
syscall
li $v0,4
la $a0,nl
syscall
li $v0,1
lw $a0,x+4
syscall
li $v0,4
la $a0,nl
syscall
li $v0,1
lw $a0,x+8
syscall
li $v0,4
la $a0,nl
syscall
li $v0,10
syscall
##########
```

va afisa (pe linii diferite):

```
1
2
3
```

si nu (pe aceeasi linie):

```
123
```

Exemplu: rezolvarea (interactiva a) ecuatiei de grad <= 1:
~~~~~~~~

```
# a*x+b=0
.data
coefa: .space 4  # a
coefb: .space 4  # b
x: .space 4      # va contine solutia x
invitatie_a:  .asciiz "Dati a: "
invitatie_b:  .asciiz "Dati b: "
solutia_este: .asciiz "Solutia este: "
incompatibilitate: .asciiz "Ecuatia nu are solutii.\n"
nedeterminare:     .asciiz "Orice numar real este solutie.\n"
nl: .asciiz "\n"
.text
main:
### citim coeficientul a
li $v0,4
la $a0,invitatie_a
syscall             # Afisam: Dati a:
li $v0,6
syscall             # Citim un float
s.s $f0,coefa
### citim coeficientul b
li $v0,4
la $a0,invitatie_b
syscall             # Afisam: Dati b:
li $v0,6
syscall             # Citim un float
s.s $f0,coefb
### rezolvam ecuatia
li.s $f0,0.0     # $f0 <- 0.0
l.s $f1,coefa    # $f1 <- 2.0 (coefa)
l.s $f2,coefb    # $f2 <- 9.5 (coefb)
c.eq.s $f1, $f0  # daca $f1=0 setam flag-ul 0 la valoarea 1
bc1t et1         # daca flag-ul 0 are val. 1, salt la et. "et1"
 # cazul coefa nenul
neg.s $f2,$f2      # $f2 <- - coefb
div.s $f2,$f2,$f1  # $f2 <- - coefb/coefa
s.s $f2,x          # scriem sol. la adr. x
li $v0,4
la $a0,solutia_este
syscall             # Afisam: Solutia este:
li $v0,2
l.s $f12,x
syscall             # Afisam valoarea lui x
li $v0,4
la $a0,nl
syscall             # Trecem la linie noua
j sfarsit
 # cazul coefa nul
et1:
c.eq.s $f2, $f0  # daca $f2=0 setam flag-ul 0 la valoarea 1
bc1t et2         # daca flag-ul 0 are val. 1, salt la et. "et2"
 # cazul coefa nul, coefb nenul
li $v0,4
la $a0,incompatibilitate
syscall             # Afisam: Ecuatia nu are solutii.
j sfarsit
 # cazul coefa nul, coefb nul
et2:
li $v0,4
la $a0,nedeterminare
syscall             # Afisam: Orice numar real este solutie.
```
~~~~~~~~

```
sfarsit:
li $v0,10
syscall
##########
```

La rularea continua (nu pas cu pas) vom vedea de exemplu pe ecran:

```
Dati a: 2
Dati b: 9.5
Solutia este: -4.75000000
```

sau:

```
Dati a: 0
Dati b: 9.5
Ecuatia nu are solutii.
```

sau:

```
Dati a: 0
Dati b: 0
Orice numar real este solutie.
```

Exemplu: cream si afisam o lista simplu inlantuita.
~~~~~~~~

Incercam sa translatam urmatorul program C:

```
struct nod{int inf; struct nod *leg;} *cap; // retine adr. capului listei
int t1, t0;                                 // nr. de elemente, indice
struct nod *t3;                             // adr. elementului curent
struct nod **t2;  // adr. locatiei ce retine adr. elem. curent (t3 sau cap)
int v0,a0;

printf("Nr. de elemente ale listei: "); scanf("%d",&t1);
printf("Dati elementele listei, cate unul pe linie:\n");
for(t0=0,t2=&cap; t0<t1; ++t0){
  t3 = (struct nod *)malloc(sizeof(struct nod));
     // aloc un element nou
  scanf("%d",&v0);
  t3->inf = v0;     // setez campul inf al elementului curent
  t3 -> leg = NULL; // setez campul leg al elem. curent (deocamdata e NULL)
  *t2=t3;           // leg elementul curent de cel precedent
  t2=&t3->leg;
}
printf("Am citit lista: ");
for(t3=cap;  t3;  t3=t3->leg ){
  a0=t3->inf;  printf("%d ",a0);
}
printf("\n");
```

Intrucat in MIPS nu putem defini structuri, vom simula structura sub forma
 unui vector de 2 word-uri; primul word va contine "inf", al doilea "leg".

```
.data
cap: .space 4  # capul listei; retine adr. primului element
.text
main:
 # citim nr. de elemente ale listei
li $v0, 4  # print string
la $a0,nr_elemente
syscall
li $v0,5   # read int
syscall
```
~~~~~~~~

```
move $t1,$v0  # $t1 va retine nr. de elemente ale listei
 # alocam si citim elementele listei
li $v0,4   # print string
la $a0,dati_elementele
syscall
li $t0,0      # $t0 va fi numarul de ordine al elementului curent
la $t2,cap    # $t2 va fi adresa locului unde se va stoca adresa urmatorului
element de lista
citesc:
bge $t0,$t1,citit
  # aloc un element (2 word, unul pt. valoare, unul pt. adr. urmatorului)
  li $v0,9   # sbrk
  li $a0,8
  syscall
  move $t3,$v0
  # citim valoarea elementului
  li $v0,5
  syscall
  sw $v0,0($t3)
  sw $zero,4($t3)  # deocamdata in campul de adr. urm. elem. pun nul
  sw $t3,($t2)
  addu $t2,$t3,4
  addiu $t0,1
  b citesc
citit:
 # afisam elementele listei pana intalnim terminatorul nul
li $v0,4   # print string
la $a0,am_citit
syscall
lw $t3,cap
scriu:
beq $t3,$zero,scris
  li $v0,1   # print int
  lw $a0,0($t3)
  syscall
  li $v0,4   # print string
  la $a0,blank
  syscall
  lw $t3,4($t3)
  b scriu
scris:
li $v0,4   # print string
la $a0,nl
syscall
li $v0,10
syscall
.data
nr_elemente:     .asciiz "Nr. de elemente ale listei: "
dati_elementele: .ascii  "Dati elementele listei, "
                 .asciiz "cate unul pe linie:\n"
am_citit:        .asciiz "Am citit lista: "
blank:           .asciiz " "
nl:              .asciiz "\n"
##########
```

La rularea continua (nu pas cu pas) vom vedea de exemplu pe ecran:

```
Nr. de elemente ale listei: 5
Dati elementele listei, cate unul pe linie:
2
1
2
3
4
```

Am citit lista: 2 1 2 3 4

Comentarii:
- pentru a face textul mai clar, am impartit sectiunea de date (.data) in doua parti, o parte pusa la inceput si care contine variabilele folosite in procesare, o parte pusa la sfarsit si care contine mesajele afisate la consola; aceasta impartire nu afecteaza modul de functionare a programului, el se executa ca si cand am fi avut o singura sectiune .data la inceput si continand toate variabilele si mesajele;
- documentatia MIPS mentionata la bibliografie nu spune ce se intampla daca sbrk nu poate aloca atata memorie cat am cerut si nu spune cum se dezaloca memoria alocata cu sbrk; surse de pe Internet spun ca in SPIM nu s-a implementat functia de dezalocare;
  de aceea nici in programul nostru nu testam succesul alocarii unui nou element si nici nu dezalocam lista la sfarsit.

F. Exercitii:
==========

Programele pentru care nu s-a cerut implementarea I/O de la consola vor avea datele initializate prin program.
Programele marcate cu (*) se vor realiza in clasa.

II.1) (2.5 puncte) (*)
Program pentru calcularea celui de-al n-lea termen al sirului lui Fibonacci (t1:=1, t2:=1, tn:=t(n-1)+t(n-2), pt. orice n>=3). Numarul n este dat printr-o variabila declarata cu initializare in program. In plus, pentru calcularea termenilor se vor folosi maxim trei alte variabile x,y,z declarate in program si nu se va folosi stiva. In final valoarea ceruta va fi stocata in z. Variabilele vor fi de tip word.

II.2) (2.5 puncte) (*)
La fel ca la problema II.1, dar pentru calcularea termenilor se va folosi stiva. In program vor fi declarate doar variabilele n (cu initializare) si z. Variabila n va fi folosita doar pentru a da si eventual numara iteratiile iar variabila z doar pentru a pune in ea la sfarsit termenul cerut (toti termenii intermediari vor fi stocati in stiva).

II.3) (3 puncte) (*)
Program care calculeaza suma divizorilor unui numar natural. Numarul este dat intr-o variabila n de tip word declarata cu initializare in program; suma va fi stocata in final intr-o variabila s de tip word.

II.4) (3 puncte)
Program care verifica daca un numar natural este prim. Numarul este dat intr-o variabila n de tip word declarata cu initializare in program; raspunsul va fi stocat intr-o variabila x de tip byte sub forma 0=neprim, 1=prim.

II.5) (1 punct daca a fost facuta problema II.4 sau 4 puncte altfel) (*)
Program care verifica daca un numar natural este prim. Numarul este citit interactiv de la consola sub forma:
  Dati numarul: 5
iar rezultatul este afisat la consola sub forma:
  Numarul 5 este prim.
La citire 5 este ce tastam noi; la afisare in loc de 5 se va afisa numarul citit; daca nu e prim mesajul se va termina "... nu e prim".

II.6) (3 puncte) (*)
Program care calculeaza suma cifrelor in baza 10 ale unui numar natural. Numarul este dat intr-o variabila n de tip word declarata cu initializare in program; suma va fi stocata in final intr-o variabila s de tip word.

II.7) (1 punct) (*)

Citire/afisare interactiva a unui vector de nr. naturale.
Dialogul se va desfasura sub forma:
dati vactorul:
n=3
v[0]=1
v[1]=5
v[2]=3
am citit vectorul: 1, 5, 3
(ce e in dreapta lui "=" am tastat noi, restul e afisat de program).

II.8) (1.5 puncte)
Citire/afisare interactiva a unei matrici (intr-o forma analoaga celei de la problema II.1).

II.9) (2.5 puncte) (*)
Program care calculeaza factorialul unui numar natural. Numarul este dat intr-o variabila n de tip word declarata cu initializare in program; factorialul va fi stocata in final intr-o variabila s de tip word.

II.10) (2.5 puncte)(*)
Program care calculeaza 1+2+...+n, unde n este o variabila word declarata cu initializare (numar natural) in program. Suma va fi stocata in final intr-o variabila s de tip word.

II.11) (2 puncte) (*)
Program care calculeaza diferenta a doua numere intregi multioctet folosind un ciclu. Numerele vor fi date prin cod, sub forma a doua variabile x,y de tip byte initializate cu cate un sir de 5 octeti - ele se vor scadea byte cu byte, cate o pereche la fiecare iteratie, cu imprumut corespunzator pentru perechea urmatoare. Pentru stocarea rezultatului se va declara o variabila z de tip .space 5. Numarul de bytes (5) va fi dat intr-o variabila n declarata cu initializare si va fi luat de acolo.

II.12) (cate 10 puncte pentru fiecare din cele doua operatii)
Ca la problema II.11, dar cu inmultire si impartire (doua programe).

II.13) (4 puncte) (*)
Program pentru evaluarea unei expresii oarecare in forma postfixata (forma postfixata pleaca de la reprezentarea operatiei ca functie cu numele functiei la sfarsit x+y = (x,y)+ si eliminarea parantezelor si virgulei; astfel (x+y)*z = xy+z* iar x*z+y*z = xz*yz*+).
Expresia are ca operanzi numere byte si ca operatori "+" si "-" binari.
Ea va fi data sub forma unei variabile .byte initializata la declarare cu un sir de octeti ce contine expresia si se termina cu caracterul nul, de exemplu expresia ((10 + 20) - (10 + 15)) - (4 - 3) va fi data prin:

```
e: .byte 10, 20, '+', 10, 15, '+', '-', 4, 3, '-', '-', 0
```

Algoritmul de evaluare este urmatorul:
(0) Notam cu i poz. curenta in e si cu e(i) byte-ul de la aceasta pozitie;
(a) i:=0
(b) Daca e(i)=0 salt la (f)
(c) Daca e(i)<>'+','-' (adica e operand) atunci push e(i)
si salt la (e)
(d) Daca e(i)='+' atunci pop doua valori, calc. suma lor, push rezultatul
si salt la (e)
Daca e(i)='-' atunci pop doua valori, calc. dif. lor, push rezultatul
si salt la (e)
(e) i:=i+1 si salt la (b)
(f) Pop o valoare (este rezultatul final al expresiei) si o salvez in x.

II.14) (2 puncte daca s-a facut problema II.14 sau 5 puncte daca nu) (*)
Ca la problema II.13, cu urmatoarele modificari:

Expresia poate contine doar operanzi de o cifra, este citita de la consola ca un string (read string) si stocata intr-o variabila e initializata cu .space 100 (in memorie ea va ajunge ca un string asciiz). Dupa citire, stringul va fi transformat a.i. operanzii sa fie stocati sub forma valorii lor matematice, nu a caracterelor care le reprezinta. In acest scop din fiecare byte avand valori de la 48 la 57 (i.e. din fiecare carecter '0', ...,'9') vom scadea 48. Apoi vom inlocui caracterul newline de la sfarsit cu un caracter nul.
De exemplu expresia ((1 + 2) - (1 + 5)) - (4 - 3) va fi introdusa de la tastatura sub forma stringului: 12+15+-43-- iar in memorie sa va stoca sub forma vectorului de bytes: 49, 50, 43, 49, 53, 43, 45, 52, 51, 45, 45, 10, 0 (avand in vedere ca codurile ASCII zecimale ale caracterelor implicate sunt: '1'->49, '2'->50, '3'->51, '4'->52, '5'->53, '+'->43, '-'->45, newline->10, nul->0). Dupa transformare, ea va ajunge sub forma: 1, 2, 43, 1, 5, 43, 45, 4, 3, 45, 45, 0, 0.
Rezultatul final al expresiei va fi afisat pe consola (print int).

II.15) (2.5 puncte) (*)
Program care calculeaza suma bitilor (numarul bitilor egali cu 1) din reprezentarea interna a unui numar natural. Numarul este dat intr-o variabila n de tip word declarata cu initializare in program; suma va fi stocata in final intr-o variabila s de tip word. Se vor folosi op. de shiftare si op. logice pentru a muta/selecta bitii.
Ex: nr. 11001000 00001111 00000011 00001101 --> suma bitilor = 12

II.16) (2.5 puncte)
Program care construieste imaginea in oglinda a configuratiei din locatia unei variabile word declarata cu initializare in program; imaginea se va construi in aceeasi locatie. Se vor folosi op. de shiftare, rotire, logice.
Ex: nr. 11001000 00001111 00000000 00000001 -->
10000000 00000000 11110000 00010011

II.17) (3 puncte) (*)
Program care roteste cu 1 la stanga in mod uniform bitii dintr-un sir de octeti declarat si initializat in program ca vector de .byte (bitul semnificativ care iese din octetul aflat la adresa n sa intre in bitul zero al octetului aflat la adresa n+1).
rang 7 0 7 0 7 0
Exemplu: 10000001 10000010 11000100 --> 00000011 00000101 10001001
adr.1 2 3

II.18) (3 puncte) (*)
Program care realizeaza adunarea a doua numere naturale word a+b (32 biti) folosind op.logice si shiftari, conform urmatorului algoritm:
(a) fac c := a xor b si obtin cele 32 cifre ale sumei daca n-am tine cont de transporturile de pe fiecare pozitie;
(b) fac d := a and b si obtin transporturile de la cele 32 poz. ale sumei (acestea trebuie adunate la pozitii shiftate la stg. cu 1, dar pot aparea noi transporturi)
(c) daca d=0...0, STOP (avem c=a+b)
(e) fac e := d shiftat la stg cu 1
(f) fac a:=c, b:=e si salt la (a)
Pentru simplitate putem considera a=$t0, b=$t1, suma=$t2; vom folosi cat mai putine locatii suplimentare (c, d, e), iar acestea pot fi registrii, din memorie sau din stiva.

II.19) (3 puncte) (*)
Program care realizeaza inmultirea a doua numere naturale word a*b (fara semn, 32 biti) folosind shiftari si adunari, conform urmatorului algoritm (care se bazeaza pe scrierea lui a ca polinom in puterile lui 2 si apl. distributivitatii, de ex. 10 * b = (2^3 + 2^1)*b = b*2^3 + b* 2^1):
(a) fac c := 0
(b) daca a=0, STOP (avem c=a*b)

(c) daca a impar (a and 0x1 != 0) fac c := c+b
(d) fac a:=a shiftat la dr. cu 1 (i.e. impart la 2),
fac b:=b shiftat la stg. cu 1 (i.e. inmultesc cu 2),
si salt la (b)

Pentru simplitate putem considera a=$t0, b=$t1, produsul=$t2, si ca jumatatea superioara a lui $t0, $t1 este 0 (pentru ca rez. sa incapa in 32 biti); vom folosi cat mai putine locatii suplimentare, iar acestea pot fi registrii, din memorie sau din stiva.

II.20) (2 puncte) (*)
Program care determina calculeaza numarul elementelor nule ale unui vector de bytes. Vectorul va fi dat sub forma unei variabile initializata la declarare cu un sir de bytes, iar lungimea lui printr-o variabila initializata de asemenea la declarare.

II.21) (2 puncte)
Program care calculeaza produsul scalar a doi vector de bytes (vectorii sunt dati sub forma a doua variabile initializate la declarare cu cate un sir de bytes, iar lungimea lor printr-o variabila initializata de asemenea la declarare).

II.22) (5 puncte) (*)
Program pentru sortarea unui vector de word. Vectorul va fi dat sub forma unei variabile initializata la declarare cu un sir de word-uri, iar lungimea lui printr-o variabila initializata de asemenea la declarare. Vectorul sortat se va construi in aceeasi locatie ca vectorul sursa.

II.23) (5 puncte)
Program care determina elementele distincte dintr-un vector de word si le pune intr-un nou vector. Vectorul sursa va fi dat sub forma unei variabile initializata la declarare cu un sir de word-uri, iar lungimea lui printr-o variabila initializata de asemenea la declarare. Pentru vectorul rezultat se va declara o variabila .space (urmata de suficienti bytes neinitializati).
Ex: 4, 2, 1, 2, 2, 1, 3 --> 4, 2, 1, 3

II.24) (5 puncte)
Program care caluleaza combinari de n luate cate k, folosind triunghiul lui Pascal (construit cu ajutorul unui singur vector de n+1 locatii in care se vor genera succesiv liniile din triunghi).
Numerele n si K sunt date sub forma a doua variabile declarate cu initializare, iar pentru vectorul de lucru se va declara o variabila urmata de un numar rezonabil de octeti neinitailizati.
Toate numerele vor fi de tip word.

II.25) (3 puncte)
Program care determina valoare unui polinom intr-un punct, folosind schema lui Horner. Vectorul coeficientilor se va da sub forma unei variabile byte initializata la declarare cu un sir de bytes, gradul si punctul se vor da sub forma unor variabile byte declarate cu initializare.

II.26) (15 puncte)
Program de inmultire a doua matrici liniarizate. Matricile sursa se dau sub forma unor variabile initializate la declarare cu siruri de word-uri, dimensiunile lor se dau sub forma a trei variabile byte declarate cu initializare, pentru matricea rezultat se va declara o variabila urmata de un numar corespunzator de bytes neinitializati.

II.27) (8 puncte)
Program care calculeaza suma elementelor maxime ale coloanelor unei matrici liniarizate (din fiecare coloana se ia cate un singur element maxim). Matricea se da sub forma unei variabile initializate la declarare cu un sir de word-uri, dimensiunile ei se dau sub forma a doua variabile word declarate cu initializare.

Ex: 1 2 3
    8 1 5 --> 8 + 2 + 5 = 15
    1 2 4

II.28) (5 puncte) (*)
Program care simuleaza functia "strncmp" din limbajul C.
Operatia se va aplica unor stringuri declarate cu initializare in program (si terminate cu 0 - deci asciiz) iar raspunsul va fi stocat intr-o variabila x de tip word sub forma: 1=mai mic, 0=egal, 2=mai mare.

II.29) (5 puncte)
Program care numara toate aparitiile unui sir ca subcuvant in alt sir.
Operatia se va aplica unor stringuri declarate cu initializare in program (si terminat cu 0) iar raspunsul va fi stocat intr-o var. n de tip word.

II.30) (4 puncte) (*)
Program care construieste imaginea in oglinda a unui sir (imaginea se va construi in aceeasi locatie). Operatia se va aplica unui string citit de la consola (in memorie se va inlocui eventualul newline de la sfarsit cu 0) iar stringul rezultat se va afisa la consola.
Ex: "abadc" --> "cdaba"

II.31 (8 puncte) (*)
Program de rezolvare a ecuatiei de grad <=2 cu coeficienti reali (.single): a*x^2 + b*x + c = 0, a,b,c reali
Coeficientii se citesc de la consola, rezultatele (solutii sau mesaje) se scriu la consola.
Se vor trata toate cazurile: a=b=c=0, a=b=0 si c!=0, a=0 si b!=0, a!=0 si delta<0, a!=0 si delta=0, a!=0 si delta>0.

II.32 (6 puncte) (*)
Program de rezolvare a problemei turnurilor din Hanoi.
Enuntul problemei:
avem trei pari notati 1, 2, 3;
pe parul 1 avem un turn de n discuri cu diametre descrescand de la baza spre varf;
trebuie sa mutam turnul de discuri de pe parul 1 pe parul 3, mutand cate un singur disc o data de pe un par pe altul, avand grija sa nu punem niciodata un disc mai mare peste un disc mai mic;
Evident, problema se poate reformula echivalent pentru a muta turnul de discuri de pe oricare din parii 1,2,3 pe oricare din parii ramasi, folosind parul ramas ca par intermediar.
Rezolvarea problemei (formulata recursiv):
notam o mutare sub forma "i -> j" inseamnand ca am mutat discul superior de pe parul i pe parul j;
notam h(n,i,j,k) succesiunea mutarilor necesare pentru a rezolva problema mutarii unui turn de n discuri de pe parul i pe parul k, folosind parul j ca par intermediar (i,j,k sunt numerele 1,2,3 intr-o ordine oarecare);
atunci: h(1,i,j,k) = "i -> k"
        h(n,i,j,k) = h(n-1,i,k,j), h(1,i,j,k), h(n-1,j,k,i) (pentru n>1)
(evident, mutarile necesitate de h(n-1,i,k,j) si h(n-1,j,k,i) nu sunt stanjenite de prezenta celui de-al n-lea disc pe parul i, deoarece are diametrul mai mare ca celelalte discuri).
Rezolvarea se poate programa usor folosind proceduri recursive, deoarece algoritmul este in esenta recursiv.
Programul va emula aceasta recursie gestionand explicit stiva, conform urmatorului algoritm:
(0) push 0 (un terminator, ca sa stim cand s-a golit stiva)
(a) push nn, 1, 2, 3 (nn este nr. initial de discuri, citit de la consola)
(b) daca in varful stivei este 0, stop
(c) pop k, j, i, n
(d) daca n=1 afisaza la consola "i -> k" (in loc de i, k se vor afisa numerele respective) si salt la (b)
(e) push n-1, j, k, i

```
      push 1, i, j, k
      push n-1, i, k, j
      salt la (b)
 Toate numerele sunt word.

II.33) (6 puncte)  (*)
 In program declar etichetele word: v initializat cu un vector de valori, n
  initializat cu numarul de elemente din vectorul anterior, x si y
  initializate cu cate un word, w intializat cu space.
 Programul creaza o lista simplu inlantita de elemente alocate dinamic
  (sbrk) continand elementele vectorului (si adresa nula la sfarsit, ca
  terminator), apoi insereaza (ca un nou element alocat dinamic) valoarea
  din y dupa elementul din lista avand valoarea din x, apoi scrie pe rand
  elementele noii liste in vectorul w (lista se parcurge pana la intalnirea
  adresei nule), apoi afisaza vectorul w la consola (pe un singur rand, cu
  elementele separate prin blank).

Bibliografie:
~~~~~~~~~~~~~
```
1."Organizarea si proiectarea calculatoarelor - interfata hardware/software",
  John L. Hennessy, David A. Patterson, ed. All, 2002,
  anexa A

DANIEL DRAGULICI
octombrie - noiembrie, 2006

Limbajul MIPS - Lectia 3 - Proceduri si Macro-uri:
~~~~~~

0. Recapitulare:
================

Registrii de uz general sunt:

Nume	Numar	Rol
$zero	$0	are mereu valoarea 0
$at	$1	rezervat pentru asamblor
$v0, $v1	$2,$3	val. prod. de o expr.sau ret. de o fct.
$a0 - $a3	$4 - $7	parametri actuali
$t0 - $t7,$t8,$t9	$8 - $15,$24,$25	val.temporare (nerestaurate de apeluri)
$s0 - $s7	$16 - $23	val.temporare (restaurate de apeluri)
$k0, $k1	$26, $27	rezervat pentru kernel
$gp	$28	pointer global
$sp	$29	pointeaza varful stivei
$fp	$30	pointeaza cadrul curent in stiva
$ra	$31	contine adresa de intoarcere din apelul de subrutina curent

 In implementarea si utilizarea subrutinelor sunt folositi mai ales registrii $a0 - $a3 (pentru transmiterea parametrilor actuali), $v0, $v1 (pentru transmiterea valorii returnate, in cazul functiilor), $sp (pentru gestionarea stivei - pointeaza in permanenta varful stivei), $fp (pointeaza in stiva cadrul apelului curent de subrutina), $ra (contine adresa de intoarcere din apelul curent de subrutina).

 Registrul PC contine mereu adresa instructiunii care urmeaza sa se execute; el este consultat/modificat indirect de instr. ca "jal", "jr", etc.

 Stiva este o zona de memorie folosita pentru stocarea de valori temporare; in particular ea este folosita la gestionarea apelurilor de subrutina (pentru stocarea temporara a unor valori legate de aceste apeluri).
 Stiva este gestionata in maniera FIFO, datele fiind incarcate/descarcate la un acelasi capat, numit varful stivei.
 Stiva creste spre adrese mici si scade spre adrese mari, iar registrul $sp are drept rol sa retina in permanenta adresa varfului stivei (a octetului din varful stivei).
 Astfel, putem incarca (push) un word din $t0 in stiva cu secventa:

```
 subu $sp,4
 sw $t0,0($sp)
```

si putem descarca (pop) word-ul din varful stivei in $t0 cu secventa:

```
 lw $t0,0($sp)
 addu $sp,4
```

 Putem incarca/descarca mai multe date in/din stiva, dar trebuie sa scadem/crestem corespunzator $sp si, evident, sa descarcam datele in ordinea inversa in care le-am incarcat (altfel nu vom recupera valorile corecte).

 In implementarea si utilizarea subrutinelor sunt folosite urmatoarele instructiuni (care au corespondent direct in limbaj masina, pe un word, nu sunt pseudoinstructiuni:

```
bltzal/bgezal rs, eticheta

## ramificare si legatura la mai mic strict / mai mare sau egal ca 0;
## efectueaza: daca rs < / >= 0
```
~~~~~~

```
#              atunci $ra <- adr. instr. urm. si apoi salt la eticheta;
#  adica: daca rs < / >= 0
#         atunci {$ra <- PC + 4; PC <- eticheta;}

jal eticheta

## salt si legatura;
## efectueaza: $ra <- adr. instr. urm. si apoi salt la eticheta;
#  adica: $ra <- PC + 4; PC <- eticheta;

jalr rd, rs

## salt si legatura in registru;
## efectueaza: rd <- adr. instr. urm. si apoi salt la adr. din rs;
#  adica: rd <- PC + 4; PC <- eticheta;
## rd poate lipsi si atunci se considera $ra ($31);

jr rs

## salt la registru;
## efectueaza: salt la adresa din rs;
#  adica: PC <- rs;
```

1. Proceduri:
==========

Daca o prelucrare trebuie facuta de mai multe ori in acelasi fel (eventual cu alte date), in loc sa rescriem grupul respectiv de instructiuni de mai multe ori in program putem sa-l incapsulam intr-o subrutina si de fiecare data cand avem nevoie de el sa apelam subrutina (eventual cu noile date transmise ca parametri).

Practic, se incarca eventualii parametri intr-un loc accesibil subrutinei (registri, memorie, stiva), se transfera executia la prima instructiune a subrutinei (apel), se executa subrutina, ocazie cu care aceasta poate plasa o valoare rezultata (valoare de retur) intr-un loc accesibil codului apelant (registri, memorie, stiva), se transfera executia la instructiunea urmatoare celei care a provocat apelul (revenire sau return) si se continua executarea codului apelant.

Incapsularea prelucrarilor in subrutine permite atat micsorarea dimensiunii codului rezultat cat si o implem. mai usoara a unor alg.complicati - odata ce am creat o subrutina ce rezolva corect o anumita problema putem sa o apelam de cate ori vrem fara a ne mai aminti exact ce prelucrari se fac in ea - e suficient sa stim doar cum se apeleaza si ce problema rezolva. De asemenea, intr-un program impartit in subrutine putem localiza/remedia mai usor erorile de programare.

In limbalele de programare uzuale subrutinele pot fi proceduri (la care sunt importante efectele laterale rezultate, cum ar fi modificarea unor variabile globale) si functii (la care este importanta valoarea returnata) - ele se scriu si se apeleaza diferit. In MIPS exista doar proceduri (deci un singur mod de scriere si apelare a subrutinelor), dar acestea pot simula si comportamentul de procedura si pe cel functie.

1a) Definire, apelare, revenire:

In MIPS procedurile nu au un mod specific de definire. Ele sunt simple blocuri de cod apelate insa intr-o maniera specifica - deci caracterul de procedura il da modul de utilizare a blocului, implementat explicit de programator.

Practic, utilizarea in maniera procedurala a unui bloc se face astfel:

- intr-un context apelant se transfera executia la prima instructiune a blocului folosind o instructiune de tip "bltzal", "bgezal", "jal", "jalr"; in felul acesta inainte de salt se va salva intr-un registru, de regula $ra, adresa instructiunii urmatoare celei care a provocat transferul - este

adresa de retur (sau de revenire), de la care trebuie sa se continue executia dupa terminarea procedurii;
- in blocul apelat ca procedura trebuie sa se execute la un moment dat o instructiune de forma "jr reg", unde "reg" este registrul in care s-a salvat adresa de retur (de regula $ra) - in felul acesta se revine in contextul apelant si se continua executia cu instructiunea urmatoare celei care a provocat apelul.

Constatam ca un bloc de cod poate fi apelat in maniera procedurala daca indeplineste urmatoarele conditii:
- prima sa instructiune este etichetata (ca sa putem specifica tinta saltului la "bltzal", "bgezal", "jal", "jalr"); aceasta eticheta va fi considerata numele procedurii (deci numele procedurilor, asemeni numelor variabilelor, vor fi simple etichete, carora compilatorul le asociaza ca semnificatie adrese de memorie);
- in bloc exista instructiuni de forma "jr reg", unde "reg" este registrul in care se salveaza adresa de retur (de regula $ra), puse a.i. indiferent care ar fi traseul executiei prin bloc pana la urma sa se execute una din ele (ca sa putem reveni din orice apel).

Exemplu: procedura care afisaza un cap de linie (trece la linie noua) - este
~~~~~~~~ utila a se apela dupa afisarea altor date (de ex. numere) deoarece nu se trece automat la linie noua.

```
.data
x: .word 1
y: .float 2.0
nl: .asciiz "\n"
.text
main:
  # afisam x si trecem la linie noua (printf("%d\n",x))
li $v0, 1   # print int
lw $a0, x   # intregul de afisat
syscall     # afisaza: 1
jal newline # apelul procedurii "newline" de trecere la linie noua
  # afisam y si trecem la linie noua (printf("%d\n",y))
li $v0, 2   # print float
l.s $f12, y # single-ul de afisat
syscall     # afisaza: 2.00000000
jal newline # apelul procedurii "newline" de trecere la linie noua
  # incheiem programul
li $v0,10
syscall
  # procedura "newline"
newline:
li $v0, 4  # print string
la $a0, nl # adresa stringului de afisat (asciiz)
syscall    # afisaza '\n', adica trece la linie noua
jr $ra     # revenire din apel
##########
```

Comentarii:

- La rulare programul va afisa:

```
1
2.00000000
```

- Instructiunile din program se executa in ordinea urmatoare:

```
 li $v0, 1
 lw $a0, x
 syscall
 jal newline # adica $ra <- adr.instr."li $v0,2" si salt la instr."li $v0,4"
 li $v0, 4
 la $a0, nl
```
~~~~~~~~

```
 syscall
 jr $ra     # adica salt la instr. "li $v0,2"
 li $v0, 2
 l.s $f12, y
 syscall
 jal newline # adica $ra <- adr.instr."li $v0,10" si salt la instr."li $v0,4"
 li $v0, 4
 la $a0, nl
 syscall
 jr $ra     # adica salt la instr. "li $v0,10"
 li $v0,10
 syscall
```

- Avand in vedere ca stringul "nl" este folosit doar in procedura "newline", puteam scrie programul mai sugestiv astfel:

```
 .data
 x: .word 1
 y: .float 2.0
 .text
 main:
  ...
 li $v0,10
 syscall
   # procedura "newline"
 newline:
 li $v0, 4
 la $a0, nl
 syscall
 jr $ra
 .data
 nl: .asciiz "\n"
```

atentie insa ca stringul "nl" se va aloca tot in zona de date statice a programului, ca in primul caz.

- In program nu a fost nevoie sa folosim stiva (deci unele proceduri pot fi implementate si fara a folosi stiva).

Corpul unei proceduri poate fi pus oriunde in program (binenteles intr-o zona aflata sub incidenta lui ".text").
Putem sa il punem in interiorul blocului care este programul principal, dar atunci cand se vor executa normal instructiunile din program se vor executa instructiunile de dinaintea corpului procedurii, apoi cele din corpul ei, apoi eventual cele de dupa corpul ei - deci corpul se va executa si fara sa fi facut vreum "jal", "jalr", etc. De aceea, daca dorim ca corpul procedurii sa fie executat doar ca urmare a unui apel (de tip "jal", "jalr", etc.), trebuie sa-l includem intre un "j eticheta" si acea "eticheta", pentru ca la executia normala sa fie sarit. De exemplu:

```
.text
main:
  li $t0,1
  j et
    proc:
    li $t0,2
    jr $ra
  et:
  li $t0,3
  jal proc
  li $t0,4
li $v0,10
syscall
```

la executie $t0 va primi succesiv valorile 1, 3, 2, 4; daca nu am pune liniile "j et", "et:" iar in corpul procedurii nu ar exista "jr $ra":

```
.text
main:
  li $t0,1
    proc:
    li $t0,2
  li $t0,3
  jal proc
  li $t0,4
li $v0,10
syscall
```

atunci la executie $t0 ar primi succesiv valorile 1, 2, 3, 2, 3, 2, 3, ...; daca in plus in corpul procedurii ar exista "jr $ra", atunci la prima executare a blocului s-ar face salt la o adresa necontrolata (valoarea initiala a lui $ra) iar efectul ar fi imprevizibil.
Mai natural, putem pune corpurile procedurilor inainte de eticheta "main" sau dupa secventa "li $v0,10", "syscall", care incheie programul - astfel ele nu vor mai fi executate la executia obisnuita a programului ci doar ca urmare a unui apel (de tip "jal", "jalr", etc.). De exemplu:

```
.text                          .text
  proc:                        main:
  li $t0,2                       li $t0,1
  jr $ra                         li $t0,3
main:                            jal proc
  li $t0,1       respectiv:      li $t0,4
  li $t0,3                     li $v0,10
  jal proc                     syscall
  li $t0,4                       proc:
li $v0,10                        li $t0,2
syscall                          jr $ra
```

Revenim la procedura "newline" din primul exemplu; ea a folosit registrii $v0 si $a0 pentru a transmite parametri actuali lui syscall; la revenirea in codul apelant acesti registri nu vor mai avea valorile de dinaintea apelului; astfel, daca apelantul ar fi salvat in vreunul din ei o valoare de care ar fi avut nevoie mai tarziu, dupa apel nu ar mai fi gasit-o; de aceea, pentru a face procedura utilizabila in orice context e bine ca ea sa salveze temporar undeva - si cel mai bine este pe stiva (stiva este menita sa stocheze valori temporare) - valorile initiale ale registrilor pe care ii modifica, iar inainte de revenire sa restaureze valorile lor vechi. Cu aceste modificari procedura se scrie:

```
newline:
subu $sp,8
sw $v0,4($sp) # salvam regsitrii
sw $a0,0($sp) #   alterati de procedura
li $v0, 4  # print string
la $a0, nl # adresa stringului de afisat (asciiz)
syscall    # afisaza '\n', adica trece la linie noua
lw $a0,0($sp) # restauram registrii
lw $v0,4($sp) #   salvati
addu $sp,8
jr $ra     # revenire din apel
```

si astfel ea poate fi apelata si de catre coduri care isi salveaza inainte in $v0 sau $a0 valori pe care doresc sa le regaseasca dupa apel. Remarcam ca procedura modifica si $sp, dar la sfarsit,prin descarcarea valorilor salvate, $sp are iar valoarea de la intrarea in apel; acesta este un alt aspect de care trebuie sa avem grija - daca procedura foloseste stiva, la iesirea din apel ea nu trebuie sa lase in ea valori reziduale (si $sp deplasat), altfel

exista riscul ca apelantul sa nu regaseasca corect valori salvate de el in stiva inaintea apelului (ele vor avea alte deplasamente fata de $sp); subliniem din nou ca valorile incarcate in stiva trebuie descarcate in ordine inversa, altfel nu se vor recupera valorile corecte; in acest sens in loc de "lw $a0,0($sp)","lw $v0,4($sp)" puteam scrie "lw $v0,4($sp)","lw $a0,0($sp)", dar nu "lw $a0,4($sp)","lw $v0,0($sp)".

Salvarea si restaurarea registrilor modificati de o procedura la inceputul, respectiv sfarsitul, apelului consuma timp de calcul si spatiu pe stiva si de aceea s-au impus conventii ce ingaduie programatorilor sa foloseasca doar anumiti registri pentru a stoca valori pe care spera sa le regaseasca mai tarziu; astfel procedurile sunt obligate sa salveze/restaureze doar acesti registri.

Mai exact, conventiile MIPS cer ca procedurile sa salveze/restaureze doar registrii $s0 - $s7, $fp si $ra (evident, doar pe cei pe care ii modifica).

De aici rezulta diferenta dintre registrii $s0 - $s7 si $t0 - $t9. Toti sunt folositi pentru a stoca valori temporare, dar numai $s0 - $s7 trebuie conservati de codurile (procedurile) apelate.

Presupunem acum ca ne aflam intr-un moment in care o procedura a apelat alta procedura, care a apelat alta procedura, etc., si ne aflam in contextul executarii apelului de la un anumit nivel (ce poate fi si cel al programului principal). Atunci:

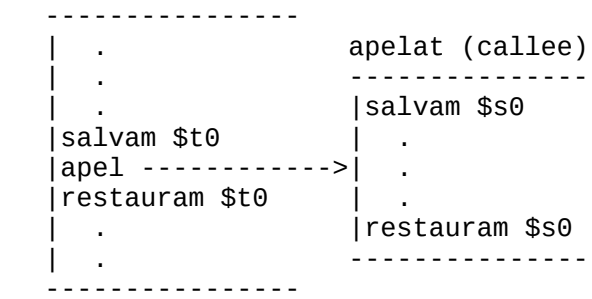
- daca in apelul curent salvam o valoare intr-unul din reg. $s0 - $s7, se garanteaza conservarea valorii pana o vom schimba tot in acelasi apel, chiar daca intre timp facem apeluri imbricate la nivel superior (reg. $s0 - $s7 sunt "preserved across call");
  aceasta se datoreaza faptului ca procedurile apelate salveaza (de ex. in stiva) la intrarea in apel reg. $s0 - $s7 folositi si ii restureaza la iesire; de aceea reg. $s0 - $s7 s.n. registri temporari salvati de apelat;
  astfel apelul curent poate stoca in $s0 - $s7 valori cu viata lunga, cum ar fi variabilele sale locale (ca variabilele "register" in limbajul C);
- daca in apelul curent salvam o valoare intr-unul din reg. $t0 - $t9, se garanteaza conservarea valorii doar daca nu facem apeluri imbricate (reg. $t0 - $t9 sunt "not preserved across call");
  de aceea,daca dorim conservarea valorii chiar daca facem apeluri imbricate, apelul curent trebuie sa-i salveze (de ex. in stiva) inainte de fiecare apel imbricat si sa-i restaureze dupa; de aceea $t0 - $t9 s.n. registri temporari salvati de apelant;
  de regula in $t0 - $t9 se stocheaza doar valori cu viata scurta, care sunt folosite (si apoi se pot pierde) la scurt timp dupa ce au fost produse, cum ar fi rezultatele partiale obtinute in procesul evaluarii expresiilor.

Mai general:

- o valoare stocata intr-unul din reg. $s0 - $s7, $fp si $ra se va conserva in urma apelurilor imbricate, deoarece acestea trebuie sa salveze/restaureze reg. respectivi pe care ii modifica - ei s.n. registri conservati de apelat (callee-saved registers);
- o valoare stocata intr-unul din reg. $a0 - $a3, $t0 - $t9 nu se va conserva in urma apelurilor imbricate decat daca apelantul ii salveaza inaintea apelului si ii restaureaza dupa apel - ei s.n.registri conservati de apelant (caller-saved registers);
- in general un apel imbricat trebuie sa lase la sfarsit in $sp aceeasi valoare pe care a avut-o la inceput, dar aceasta nu se realizeaza prin salvare/restaurare ca in cazul celorlalti registri ci rezulta din modul de administrare a stivei (apelul trebuie sa descarce din stiva toate valorile temporare incarcate de el); daca nu respectam regula asta, la revenire apelantul nu va regasi valorile salvate de el in stiva (ele vor avea alte deplasamente fata de $sp) decat daca foloseste alt instrument pentru a le referi (de exemplu $fp - a se vedea mai jos);
  ceilalti registri ori nu trebuie modificati ($zero, $at, $k0, $k1, $gp), ori nu sunt folositi pentru a stoca valori temporare - de ex. $v0, $v1 se folosesc doar pentru a recupera valoarea finala a unei expresii sau pentru a furniza valoarea returnata de o functie; in caz contrar si ei trebuie salvati de apelant.

Ilustram printr-un desen momentele cand trebuie salvat/restaurat un registru

caller-saved - de ex. $t0 - si un registru callee-saved, de ex.$s0:



Subliniem ca regulile de mai sus sunt doar conventii de scriere a programelor si nu restrictii hardware; ele pot fi incalcate, dar codul nostru risca sa nu fie compatibil cu altele; in orice caz conventiile trebuie respectate daca ceea ce scriem nu este un program complet ci doar un fragment care va fi combinat cu fragmente scrise de altii.

Exemplu: folosirea registrilor temporari salvati de apelant si apelat;
~~~~~~~~

Consideram programul C:

```
#include<stdio.h>

int s1=1,s2=1;         /* variabile globale */

void linie_s1(){
  register int i;  /* variabila locala lui "linie_s1", alocata register */
  for(i=0; i<s1; ++i) printf("*");
  printf("\n");
}

void linie_s2(){
  register int i;  /* variabila locala lui "linie_s2", alocata register */
  for(i=0; i<s2; ++i) printf("-");
  printf("\n");
}

void main(){
  register int i;  /* variabila locala lui "main", alocata register */
  for(i=0; i<3; ++i){
    s1=s1+i*i;
    linie_s1();
    s2=s2+i*i*i;
    linie_s2();
  }
}
```

la rulare acesta afisaza:

```
*
-
**
--
******
----------
```

O translatare a acestui program este:

```
.data
  s1: .word 1
```
~~~~~~~~

```
  s2: .word 1
.text
main:
  li $s0,0  # alocam i din "main" in $s0
  inceput:
  li $t0,3
  bge $s0,$t0 sfarsit
    move $t0,$s0      # in $t0 calculam i*i, apoi i*i*i
    mulo $t0,$t0,$s0  # acum, $t0 = i*i
    lw $t1,s1
    add $t1,$t1,$t0
    sw $t1,s1
    subu $sp,4     # $t0 ne trebuie si mai tarziu si nu e salvat de apelat,
    sw $t0,0($sp)  #  deci il salvam in apelant; acum stiva este: $sp:($t0 v)
    jal linie_s1
    lw $t0,0($sp)  # dupa revenire stiva este tot: $sp:($t0 v)
    addu $sp,4     #  si restauram $t0 la revenirea in apelant
    mulo $t0,$t0,$s0  # acum, $t0 = i*i*i
    lw $t1,s2
    add $t1,$t1,$t0
    sw $t1,s2
    jal linie_s2   # acum nu ne mai trebuie in apelant $t0 si nu-l mai salvam
    add $s0,1
    b inceput
  sfarsit:
li $v0,10
syscall
.data
  stea:  .asciiz "*"
  linie: .asciiz "-"
  nl:    .asciiz "\n"
.text
linie_s1:       # stiva primita la inceput este:  $sp:($t0 v)
  subu $sp,4    # salvam $s0 in apelat, deoarece il va modifica
  sw $s0,0($sp) #        acum stiva este:  $sp:($s0 v)($t0 v)
  li $s0,0  # alocam i din "linie_s1" in $s0
  inceput1:
  lw $t0,s1
  bge $s0,$t0,sfarsit1
    li $v0,4  # print string
    la $a0,stea
    syscall
    add $s0,1
    b inceput1
  sfarsit1:
  li $v0,4
  la $a0,nl
  syscall
  lw $s0,0($sp) # restauram valoarea veche a lui $s0;
  addu $sp,4    #  acum stiva este ca la inceput: $sp:($t0 v)
jr $ra
linie_s2:       # stiva primita la inceput este vida
  subu $sp,4    # salvam $s0 in apelat, deoarece il va modifica
  sw $s0,0($sp) #        acum stiva este:  $sp:($s0 v)
  li $s0,0  # alocam i din "linie_s2" in $s0
  inceput2:
  lw $t0,s2
  bge $s0,$t0,sfarsit2
    li $v0,4  # print string
    la $a0,linie
    syscall
    add $s0,1
    b inceput2
  sfarsit2:
```

```
  li $v0,4
  la $a0,nl
  syscall
  lw $s0,0($sp) # restauram valoarea veche a lui $s0;
  addu $sp,4    #  acum stiva este ca la inceput, vida
jr $ra
##########
```

Comentarii:
- registrii $s0 si $t0 au fost folositi si in programul principal (apelant) si in procedurile "linie_s1" si "linie_s2" (apelate) pentru a stoca valori; in apelant, valoarea din $s0 trebuie sa supravietuiasca ambelor apeluri imbricate, iar cea din $t0 doar primului apel; conform conventiilor MIPS, $s0 a fost salvat/restaurat in ambii apelati, iar $t0 in apelant (dar doar pentru a supravietui primului apel imbricat);
  procedurile "linie_s1" si "linie_s2" modifica de asemenea si $v0, $a0, dar apelantul nu doreste sa conserve valorile lor peste apelurile imbricate, deci nu le salveaza/restaureaza;
- chiar daca $s0 n-ar trebui sa supravietuiasca ambelor apeluri imbricate, acestea tot ar trebui sa il salveze/restaureze daca respectam conventiile MIPS, intrucat codul procedurilor respective il modifica;
  de asemenea, putem gasi o alta transcriere a programului C, in care procedurile sa foloseasca alt registru $s (de ex. $s1) pentru a-si aloca variabila i si alt registru $t (de ex. $t7) decat cei folositi in programul principal; atunci nu va mai fi nevoie sa salvam/restauram niciunul dintre $s0 si $t0, dar daca vrem sa aplicam strict conventiile MIPS (mai ales daca procedurile si programul principal sunt scrise de programatori diferiti si nu stie unui daca celalalt chiar foloseste registrii respectivi) tot trebuie sa o facem;

In final formulam pe scurt si retinem urmatoarea

CONVENTIE MIPS:
- registrii $s0 - $s7, $fp si $ra s.n. registri conservati de apelat (callee-saved registers);
- registrii $a0 - $a3, $t0 - $t9 s.n. registri conservati de apelant (caller-saved registers);
- un cod apelant trebuie sa salveze/restaureze cu ocazia fiecarui apel imbricat doar acei registri caller-saved pe care doreste sa-i conserve peste apel, iar un cod apelat trebuie sa salveze/restaureze toti registrii callee-saved pe care ii modifica (indiferent cine o apeleaza).

In comentariile incluse in exemplul de mai sus am folosit un anumit mod de a descrie continutul stivei pe care il vom folosi si in continuare. De exemplu o scriere de forma:

  $sp:(1)($t1)($t0 v)$fp:($a0 v)

inseamna ca:
- in stiva se afla succesiv, de la varf spre baza: 1, apoi valoarea lui $t1, apoi valoarea veche a lui $t0, apoi valoarea veche a lui $a0 (deci ultimul introdus este 1);
- registrul $sp pointeaza locatia ce contine pe 1 (deci varful stivei), iar registrul $fp pointeaza locatia ce contine valoarea veche a lui $a0 (deci nu am intercalat valorile lui $sp si $fp printre valorile din stiva, ci ei contin adresele unde am stocat in stiva pe 1 si pe $a0 vechi).

Notam ca instructiunile de apelare "bltzal", "bgezal", "jal", "jalr" si cea de revenire "jal" controleaza doar pe unde o va lua firul executiei, nu si pasarea parametrilor, preluarea valorilor returnate (in cazul functiilor), etc. - acestea mecanisme trebuie implementate separat in mod explicit, prin alte instructiuni.

1b) Variabile locale automatice, cadrul de apel:

O procedura poate avea variabile locale automatice (alocate temporar in stiva pe perioada apelului). Urmatorul exemplu arata un mod simplu de a lucra cu variabile locale automatice:

Exemplu: translatam urmatorul program C:
~~~~~~~~

```
#include<stdio.h>

void proc(){
  int p, q;       // variabile locale alocate automatic (pe stiva)
  register int i; // variabila locala alocata register
  p=0; q=0;
  for(i=0; i<3; ++i){
    ++p;
    q=(q+i)*(q-i)-p*i;
    printf("%d\n",q);
  }
}

void main(){
  proc();
}
```

La rulare afisaza:

```
0
-3
-1
```

Translatarea este:

```
.text
main:
  jal proc
li $v0,10
syscall
.data
  nl: .asciiz "\n"
.text
proc:           # la intrarea in procedura stiva este vida
  subu $sp,4
  sw $s0,0($sp) # salvam reg. callee-saved $s0, deoarece il vom modifica
  subu $sp,8    # rezervam spatiu pe stiva pentru locatiile lui p, q
    # convin ca p sa fie la 4($sp) si q la 0($sp);se poate conveni si invers;
    # acum stiva este $sp:(q)(p)($s0 v)
  sw $zero,4($sp)  # p=0;
  sw $zero,0($sp)  # q=0;
  li $s0,0         # aloc i in $s0; acum fac i=0;
  li $t0,3         # fixez $t0 la valoarea 3
  inceput:
  bge $s0,$t0,sfarsit
    lw $t1,4($sp)
    add $t1,1
    sw $t1,4($sp)     # ++p
      # evaluez (q+i)*(q-i)-p*i cu stiva
      # reamintim ca acum stiva este $sp:(q)(p)($s0 v)
    lw $t1,4($sp)     # evaluez p*i si incarc in stiva
    mulo $t1,$t1,$s0  #
    subu $sp,4        #
    sw $t1,0($sp)
      # acum stiva este $sp:(p*i)(q)(p)($s0 v)
```
~~~~~~~~

```
      # deci acum p este la 8($sp) iar q este la 4($sp)
    lw $t1,4($sp)     # evaluez q-i si incarc in stiva
    sub $t1,$t1,$s0   #
    subu $sp,4        #
    sw $t1,0($sp)     #
      # acum stiva este $sp:(q-i)(p*i)(q)(p)($s0 v)
      # deci acum p este la 12($sp) iar q este la 8($sp)
    lw $t1,8($sp)     # evaluez $t1=q+i
    add $t1,$t1,$s0   #
    lw $t2,0($sp)     # descarc din stiva q-i si evaluez $t1=(q+i)*(q-i)
    addu $sp,4        #
    mulo $t1,$t1,$t2  #
      # acum stiva este $sp:(p*i)(q)(p)($s0 v)
    lw $t2,0($sp)     # descarc din stiva p*i si evaluez $t1=(q+i)*(q-i)-p*i
    addu $sp,4        #
    sub $t1,$t1,$t2   #
      # acum stiva este $sp:(q)(p)($s0 v)
      # deci acum iar p este la 4($sp) iar q este la 0($sp)
    sw $t1,0($sp)     # atribui q=(q+i)*(q-i)-p*i
      # afisez q
    li $v0,1      # print int
    lw $a0,0($sp) # $a0=q
    syscall
    li $v0,4      # print string
    la $a0,nl
    syscall
    add $s0,1     # ++i
    b inceput
  sfarsit:
    # acum stiva este $sp:(q)(p)($s0 v)
  addu $sp,8    # dezaloc din stiva variabilele automatice p, q
    # acum stiva este $sp:($s0 v)
  lw $s0,0($sp) # restauram $s0
  addu $sp,4
    # inaintea iesirii din procedura stiva este iar vida, ca la inceput
jr $ra
##########

Comentarii:
- variabila register i a fost alocata intr-un registru "preserved across
 call", deoarece asa este recomandabil (sa folosim un registru destinat
 valorilor cu viata lunga);
  salvarea/restaurarea valorii vechi a lui $s0 de catre procedura nu este
 necesara, dar asa respectam conventiile MIPS (este un registru callee-saved
 pe care procedura il modifica);
- variabilele locale automatice p si q au fost alocate pe stiva la inceputul
 apelului (scazand $sp) si dezalocate la sfarsitul apelului (pentru a lasa
 stiva ca la inceput);
- pe parcursul apelului stiva a fluctuat (prin incarcarea/descarcarea
 rezultatelor partiale temporare in procesul evaluarii expresiei
 (q+i)*(q-i)-p*i); de aceea, desi locatiile lui p si q exista in acelasi loc
 pe stiva pe toata durata apelului, offset-urile lor fata de $sp au variat
 (de exemplu p a fost la 4($sp), 8($sp), 12($sp));
  acest fenomen al variatiei offset-urilor variabilelor locale automatice
 in raport cu $sp pe perioada apelului face dificila accesarea lor cu $sp,
 mai ales daca ea se face intr-un ciclu in care stiva fluctueaza de la o
 iteratie la alta - atunci nu putem indica offsetul printr-o valoare imediata
 (deci constanta) si astfel scrierea codului este mai dificila;
  solutia este folosirea altui registru pentru accesarea entitatilor
 temporare din stiva specifice unui apel si care isi pastreaza semnificatia
 si locatia pe toata perioada apelului - cum ar fi variabilele locale
 automatice, registrii callee-saved salvati si, vom vedea mai tarziu,
 parametrii actuali primiti prin stiva; conform conventiilor MIPS acest
 registru este $fp;
```

practic, la inceputul apelului (inainte de a accesa entitatile din stiva) se plaseaza $fp sa pointeze intr-un anumit loc si va fi mentinut acolo pe toata perioada apelului; in acest fel, indiferent cum va fluctua stiva pe perioada apelului, entitatile cu locatie fixa pe perioada apelului (variabile locale automatice, registri callee-saved salvati, parametri actuali) vor avea offset-uri fixe fata de $fp, care se pot determina in faza de elaborare a programului si indica in cod prin valori imediate;
per total $fp va fi folosit pentru a accesa entitatile din stiva specifice apelului (cu locatie fixa pe perioada apelului) iar $sp va fi folosit doar pentru a controla stiva (pentru a pointa varful stivei) si eventual a accesa celelalte valori incarcate temporar (cu viata scurta);
intrucat daca programam dupa o anumita disciplina se presupune ca fiecare procedura foloseste probabil $fp (si doreste conservarea valorii acestuia pe toata perioada apelului ei, chiar daca face apeluri imbricate), acest registru este si el din categoria callee-saved - trebuie salvat/restaurat de apelat daca acesta il modifica.

Exemplu: alta translatare a programului din exemplul anterior, folosind $fp:
~~~~~~~~

```
.text
main:
  jal proc
li $v0,10
syscall
.data
  nl: .asciiz "\n"
.text
proc:           # la intrarea in procedura stiva este vida
    # rezervam spatiu pe stiva pentru entitatile cu locatie fixa pe perioada
    #  apelului - acest spatiu se va numi cadru de apel (vom vedea)
  subu $sp,16
    # salvam registrii callee-saved pe care ii modificam, in cadrul de apel
  sw $fp,12($sp)
  sw $s0,8($sp)
    # convenind ca sub cei doi registri salvati sa avem q, p,
    #  acum stiva este $sp:(q)(p)($s0 v)($fp v)
  addiu $fp,$sp,12  # plasam $fp sa pointeze primul word din cadru
    # acum stiva este $sp:(q)(p)($s0 v)$fp:($fp v)
    #  deci p este la -8($fp) iar q la -12($fp) (offset-uri fixe)
    # puteam alege orice pozitie pentru a plasa sa pointeze $fp, dar am
    #  ales-o pe aceasta (primul word din cadru) pentru a ne conforma unor
    #  conventii MIPS (a se vedea mai jos)
  sw $zero,-8($fp)   # p=0;
  sw $zero,-12($fp)  # q=0;
  li $s0,0           # aloc i in $s0; acum fac i=0;
  li $t0,3           # fixez $t0 la valoarea 3
  inceput:
  bge $s0,$t0,sfarsit
    lw $t1,-8($fp)
    add $t1,1
    sw $t1,-8($fp)     # ++p
      # evaluez (q+i)*(q-i)-p*i cu stiva
      # reamintim ca acum stiva este $sp:(q)(p)($s0 v)$fp:($fp v)
    lw $t1,-8($fp)     # evaluez p*i si incarc in stiva
    mulo $t1,$t1,$s0   #
    subu $sp,4         #
    sw $t1,0($sp)
      # acum stiva este $sp:(p*i)(q)(p)($s0 v)$fp:($fp v)
      # desi acum p, q sunt la alte offset-uri fata de $sp, ele sunt la
      #  aceleasi offset-uri (-8, respectiv -12) fata de $fp
    lw $t1,-12($fp)    # evaluez q-i si incarc in stiva
    sub $t1,$t1,$s0    #
    subu $sp,4         #
```
~~~~~~~~

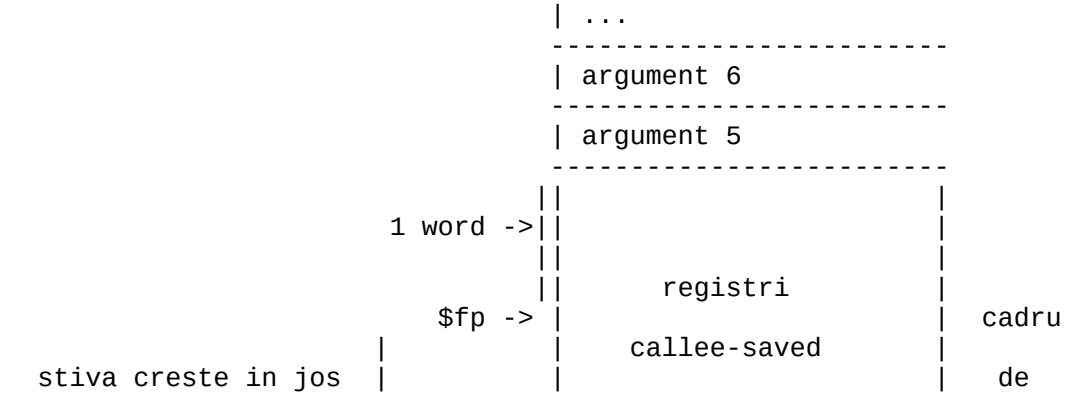
```
    sw $t1,0($sp)      #
      # acum stiva este $sp:(q-i)(p*i)(q)(p)($s0 v)$fp:($fp v)
    lw $t1,-12($fp)    # evaluez $t1=q+i
    add $t1,$t1,$s0    #
    lw $t2,0($sp)     # descarc din stiva q-i si evaluez $t1=(q+i)*(q-i)
    addu $sp,4        #
    mulo $t1,$t1,$t2  #
      # acum stiva este $sp:(p*i)(q)(p)($s0 v)$fp:($fp v)
    lw $t2,0($sp)     # descarc din stiva p*i si evaluez $t1=(q+i)*(q-i)-p*i
    addu $sp,4        #
    sub $t1,$t1,$t2   #
      # acum stiva este $sp:(q)(p)($s0 v)$fp:($fp v)
    sw $t1,-12($fp)   # atribui q=(q+i)*(q-i)-p*i
      # afisez q
    li $v0,1        # print int
    lw $a0,-12($fp) # $a0=q
    syscall
    li $v0,4        # print string
    la $a0,nl
    syscall
    add $s0,1     # ++i
    b inceput
  sfarsit:
    # acum stiva este $sp:(q)(p)($s0 v)$fp:($fp v)
    # restauram registrii callee-saved salvati la inceput, $s0 si $fp
  lw $s0,-4($fp)
  lw $fp,0($fp)   # acum nu mai pot accesa cadrul cu $fp caci are alta val.
  addu $sp,16     # descarc cadrul de apel
    # inaintea iesirii din procedura stiva este iar vida, ca la inceput
jr $ra
##########

Comentarii:
- acest exemplu evidentiaza urmatorul aspect: orice apel aloca pe stiva pe
 perioada executiei sale niste entitati temporare pe care pana in final le
 va dezaloca complet; unele entitati isi conserva locatia pe toata perioada
 apelului (ca variabilele locale automatice si registrii callee-saved
 salvati), altele apar si dispar in diverse momente ale executiei apelului,
 alocandu-se in diverse locuri in stiva (ca rezultatele partiale obtinute in
 procesul evaluarii expresiilor);
  entitatile care isi conserva locatia in acelasi loc pe toata perioada
 apelului (le vom numi entitati din prima categorie) pot fi anticipate in
 faza de elaborare a programului, inclusiv dimensiunea lor totala; celelalte
 (le vom numi entitati din a doua categorie) nu pot fi anticipate mereu in
 aceasta faza - de exemplu daca avem de evaluat cu stiva o expresie avand un
 numar de subexpresii ce depinde de o valoare citita de la consola, numarul
 de rezultate partiale incarcate/descarcate in stiva pentru evaluarea ei
 difera de la o executie la alta;
  atunci putem organiza apelul astfel:
 * la inceput rezervam spatiu pe stiva pentru toate entitatile din prima
  categorie  (scazand din $sp dimensiunea lor totala, cunoscuta in faza
  elaborarii programului); aceasta locatie se numeste cadru de apel (al
  apelului respectiv); alte denumiri: cadru de stiva, procedure call frame
  (in engleza);
 * apoi initializam parti ale cadrului de apel cu anumite valori, daca e
  cazul (de exemplu salvam aici registrii callee-saved pe care urmeaza sa-i
  modificam); in aceasta faza $sp ramane fixat si putem accesa parti ale
  cadrului de apel cu offset-uri fixe fata de $sp, care se pot determina in
  faza de elaborare a programului (si specifica in program prin valori
  imediate) deoarece dimensiunea cadrului de apel si entitatile ce trebuie
  salvate acum in el se cunosc;
 * apoi plasam $fp intr-o pozitie convenabila in raport cu cadrul de apel
  (la un offset ales si cunoscut, de asemenea, in faza de elaborare a
  programului); din acest moment putem accesa entitatile din prima categorie
```

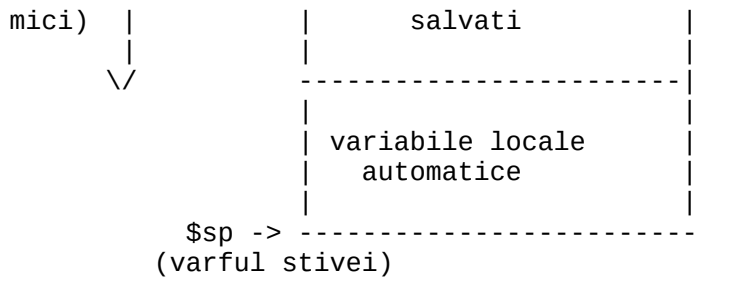
aflate in cadrul de apel folosind offset-uri constante fata de $fp (care sunt si ele cunoscute in faza de elaborare a programului si pot fi indicate in cod prin valori imediate);
de notat ca $fp este callee-saved deci, pentru ca il modificam, el trebuie salvat inainte in cadrul de apel;
* apoi efectuam prelucrarea specifica apelului, ocazie cu care in stiva se vor mai incarca/descarca entitati din a doua categorie iar stiva va fluctua in continuarea cadrului de apel (la adrese inferioare), neafectand in vreun fel entitatile din prima categorie salvate in el;
cand terminam prelucrarea specifica apelului si vrem sa iesim, trebuie sa avem grija ca $sp sa pointeze din nou cadrul de apel (sa nu ramana incarcate in stiva valori in plus);
* inainte de a iesi din apel restauram registrii callee-saved salvati (inclusiv $fp) si apoi descarcam cadrul de apel din stiva adunand la $sp dimensiunea lui (cunoscuta in faza elaborarii programului); de notat ca pana la restaurarea lui $fp putem acceasa valorile din cadrul de apel cu offset-uri fata de $fp; dupa aceea nu se mai poate, deoarece $fp are alta valoare; de aceea descarcarea cadrului din stiva (care se face dupa restaurarea lui $fp) se face adunand o valoare imediata (cunoscuta in faza elaborarii programului) la $sp si nu folosind $fp;
- in programul de mai sus am aplicat pasii de mai sus, stiind ca dimensiunea cadrului de apel trebuie sa fie 16 octeti (pentru a incapea $s0,$fp,p,q).

Cadrul de apel poate fi organizat oricum; de exemplu intai registrii callee-saved salvati si apoi variabilele locale automatice (ca in exemplul de mai sus) sau invers.
La fel, $fp poate fi plasat oriunde in raport cu cadrul de apel; de exemplu il putem plasa in extremitatea sa superioara, sau la un word distanta de ea (ca in exemplul de mai sus), sau in extremitatea sa inferioara - atunci toate entitatile din cadrul de apel vor avea offset-uri pozitive fata de $fp.
Si aici insa exista conventii; mai exact avem urmatoarele

CONVENTII MIPS:
- Cadrul de apel are cel putin 24 octeti;
- Partea cadrului aflata la adrese mari stocheaza registrii callee-saved salvati; partea aflata la adrese mici contine locatiile variabilelor locale automatice;
- $fp pointeaza primul word din cadru (aflat la adresa cea mai mare);
- crearea/distrugerea cadrului de apel se face urmand pasii "*" de mai sus (vom formula mai precis in sectiunea 1j).

Asadar, conform conventiilor de mai sus, initial $fp pointeaza primul word din cadru, $sp ultimul word din cadru (care coincide cu varful stivei) iar distanta dintre $fp si $sp este dimensiunea cadrului minus 4; practic cadrul este zona de stiva dintre adresele $sp si $fp+4.
Asa cum vom vedea mai tarziu, conventiile MIPS cer ca primii 4 parametri actuali sa fie transmisi prin registrii $a0 - $a3 iar urmatorii prin stiva; ei vor aparea in ordine deasupra cadrului de apel si vor putea fi accesati tot cu offset-uri constante fata de $fp (cunoscute in faza elaborarii programului). Desenul este:





Observatie: Cartea [1], prin anumite exemple si comentarii cam neclare, pare ca sugereaza si alte conventii MIPS, cum ar fi o anumita ordine a salvarii registrilor callee-saved in cadrul de apel si faptul ca $sp trebuie sa fie aliniat mereu la cuvant dublu (adresa multiplu de 8). Alte exemple sugereaza ca si parametrii actuali transmisi prin stiva se considera ca fac parte din cadrul de apel (?). In fine, cartea [1] mentioneaza ca compilatorul gcc genereaza cod ce foloseste $fp conform convetiilor de mai sus, in timp ce compilatorul MIPS de obicei nu foloseste $fp si genereaza cod care acceseaza cadrul de apel cu $sp (deci generarea codului este mai complicata, deoarece offset-urile entitatilor din cadrul de apel fata de $sp pot fluctua).

In cele ce urmeaza nu vom respecta decat partial conventiile enuntate; de exemplu vom dimensiona cadrul de apel cat e necesar (nu de minim 24 octeti).

1c) Transmiterea de parametri:

O procedura poate comunica cu restul programului folosind parametri (prin valoare sau referinta); acestia pot fi transmisi (pasati) prin registri, memorie sau stiva. In urmatoarele trei exemple ilustram aceste modalitati in cazul unei proceduri ce aduna doua word-uri (parametri prin valoare) si pune suma intr-o variabila word (parametru prin referinta);practic se translateaza urmatorul program C:

```
void aduna(int *r, int p, int q){
  *r=p+q;
}

int x=1, y=2, z=10, rez1, rez2;

void main(){
 aduna(&rez1,x,y);  /* rez1 devine 3 */
 aduna(&rez2,z,20); /* rez2 devine 30 */
}
```

Exemplu: transmiterea parametrilor prin registri - incarc parametrii actuali in registri si apelez procedura; ea si-i va lua de acolo;
~~~~~~~~
codul este simplu si rapid, executabilul este mic, dar parametrii nu pot fi multi si/sau mari deoarece in general avem putini reg. liberi disponibili;
conform conventiilor MIPS reg. folositi pentru transmiterea parametrilor sunt $a0 - $a3;

```
.data
  x: .word 1
  y: .word 2
  z: .word 10
  rez1: .space 4
  rez2: .space 4
.text
main:
  la $a0,rez1
  lw $a1,x
  lw $a2,y
  jal aduna
```
~~~~~~~~

```
    # acum rez1 = 3
  la $a0,rez2
  lw $a1,z
  li $a2,20
  jal aduna
    # acum rez2 = 30
li $v0,10
syscall
aduna:
  add $a1,$a1,$a2
  sw $a1,0($a0)
jr $ra
##########
```

Exemplu: transmiterea prin memorie - parametrii actuali se incarca in niste
~~~~~~~~ variabile din memorie (variabile statice) iar procedura si-i ia de
          acolo;
 codul este mai lung si mai lent(memoria se acceseaza mai lent ca registrii):

```
.data
  x: .word 1
  y: .word 2
  z: .word 10
  rez1: .space 4
  rez2: .space 4
.text
main:
  la $t0,rez1
  sw $t0,r
  lw $t0,x
  sw $t0,p
  lw $t0,y
  sw $t0,q
  jal aduna
    # acum rez1 = 3
  la $t0,rez2
  sw $t0,r
  lw $t0,z
  sw $t0,p
  li $t0,20
  sw $t0,q
  jal aduna
    # acum rez2 = 30
li $v0,10
syscall
.data
  r: .space 4 # primul parametru al procedurii
  p: .space 4 # al doilea parametru al procedurii
  q: .space 4 # al treilea parametru al procedurii
.text
aduna:
  lw $t0,p
  lw $t1,q
  add $t0,$t0,$t1
  lw $t1,r
  sw $t0,0($t1)
jr $ra
##########
```

Comentariu: am scris de doua ori ".data" si ".text" pentru sugestivitate
 (pentru a fi clar ca p, q, r sunt introduse in program pentru uzul
 procedurii); alocarea entitatilor in memorie se face insa la fel ca si cand
 le-am fi scris in continuare in aceleasi sectiuni ".data", respectiv ".text"
 (in memorie programul are o singura zona de date statice si o singura zona
~~~~~~~~

de cod):

```
.data
  x: .word 1
  y: .word 2
  z: .word 10
  rez1: .space 4
  rez2: .space 4
  r: .space 4
  p: .space 4
  q: .space 4
.text
main:
  ...
li $v0,10
syscall
aduna:
  ...
jr $ra
```

in particular p, q, r se aloca in aceeasi zona (de date statice) ca si x, y, z, rez1, rez2. Notam ca $t0, $t1 nu sunt callee-saved, deci nu a fost nevoie sa-i salvam/restauram in procedura.

Exemplu: transmiterea prin stiva - parametrii actuali se incarca in stiva
~~~~~~~~ iar procedura si-i ia de acolo (parametrii trebuie incarcati in stiva in aceeasi ordine in care ii cauta procedura);
este cel mai puternic si mai flexibil mod de a pasa parametrii; dimensiunea si viteza codului sunt de obicei intermediare intre cele obtinute in cazul transmiterii prin registri si prin memorie; structura codului este insa mai complicata:

```
.data
  x: .word 1
  y: .word 2
  z: .word 10
  rez1: .space 4
  rez2: .space 4
.text
main:
    # push y, x, adr. rez1
  subu $sp,12
  la $t0,rez1
  sw $t0,0($sp)
  lw $t0,x
  sw $t0,4($sp)
  lw $t0,y
  sw $t0,8($sp)
    # acum stiva contine $sp:(adr.rez1)(x)(y)
  jal aduna
    # acum rez1 = 3
    # stiva insa contine (ca la inceput) $sp:(adr.rez1)(x)(y) si o descarc
  addu $sp,12
    # acum stiva este vida
    # push 20, z, adr. rez2
  subu $sp,12
  la $t0,rez2
  sw $t0,0($sp)
  lw $t0,z
  sw $t0,4($sp)
  li $t0,20
  sw $t0,8($sp)
    # acum stiva contine $sp:(adr.rez2)(z)(20)
  jal aduna
```
~~~~~~~~

```
    # acum rez2 = 30
    # stiva insa contine (ca la inceput) $sp:(adr.rez2)(z)(20) si o descarc
  addu $sp,12
    # acum stiva este vida
li $v0,10
syscall
aduna:  # la intrarea in apel stiva este $sp:(r)(p)(q)
  subu $sp,4
  sw $fp,0($sp)    # salvam $fp
  addiu $fp,$sp,0  # acum stiva este $sp:$fp:($fp v)(r)(p)(q)
  lw $t0,8($fp)
  lw $t1,12($fp)
  add $t0,$t0,$t1
  lw $t1,4($fp)
  sw $t0,0($t1)
  lw $fp,0($fp) # restauram $fp
  addu $sp,4    # descarc cadrul de apel; acum stiva este iar $sp:(r)(p)(q)
jr $ra
##########
```

Comentarii:
- putem introduce in stiva parametrii actuali in alta ordine, dar atunci si in procedura trebuie sa-i accesam conform noii ordini; descarcarea lor din stiva la sfarsit poate fi facuta de apelant (dupa revenirea din apelul imbricat) sau de apelat (inainte de revenire); in exemplul nostru am ales prima varianta (conform conventiei C - a se vedea in sectiunea 1e);
- intrucat parametrii actuali sunt entitati care isi conserva locatia pe toata perioada apelului (asemeni variabilelor locale automatice si registrior callee-saved salvati), este de preferat sa-i accesam si pe ei cu $fp; de aceea, chiar daca nu avem variabile locale automatice si registri callee-saved salvati, am creat un cadru de apel (conform conventiilor MIPS descrise mai inainte), care contine insa doar $fp vechi (salvat/restaurat).

CONVENTIE MIPS: la apelarea unei proceduri primii patru parametri actuali trebuie transmisi prin registrii $a0 - $a3, iar urmatorii prin stiva (in felul acesta programele simple (cu putini parametri) se executa repede).

In exemplele care urmeaza, din considerente didactice, toti parametrii actuali vor fi transmisi prin stiva (pentru a ne obisnui cu acest mod general si flexibil de lucru).

1d) Returnarea valorilor (functii):

In limbajele de nivel inalt intalnim atat subrutine de tip procedura (cum sunt cele tratate pana acum) cat si de tip functie, care nu isi propun sa produca efecte laterale ci doar sa calculeze o anumita valoare (de ex. pe baza parametrilor) pe care s-o furnizeze la sfarsit (s-o returneze) codului apelant. In MIPS functiile se vor defini tot ca niste proceduri (avem un singur concept de subrutina), dar in care vom implementa un comportament de tip functie.
La fel ca transmiterea parametrilor, returnarea unei valori se poate face prin registri, memorie sau stiva. In continuare ilustram cele trei modalitati in cazul unei functii "f" ce primeste ca parametri doua worduri si returneaza suma patratelor lor (ignorand depasirile). Asa cum am anuntat mai inainte, toti parametrii vor fi transmisi prin stiva.

Exemplu: returnarea prin registri - este cel mai des folosita, codul este
~~~~~~~~ scurt si rapid, dar nu se pot returna valori de dimensiuni mari
(vectori, stringuri, etc.);
conform conventiilor MIPS reg. folositi pentru returnarea valorilor sunt $v0 si $v1.

Translatam urmatorul program C:
~~~~~~~~

```
int f(int p, int q){
  return p*p+q*q;
}

int x;

void main(){
  x=f(1,2);  /* x devine 5 */
}
```

Translatarea este:

```
.data
  x: .space 4
.text
main:
  subu $sp,8      # incarc parametrii actuali
  li $t0,1
  sw $t0,0($sp)
  li $t0,2
  sw $t0,4($sp)  # acum stiva este $sp:(1)(2)
  jal f
  addu $sp,8     # descarc parametrii actuali
  sw $v0,x       # recuperez valoarea returnata
    # acum x = 5
li $v0,10
syscall
f:              # la intrarea in apel stiva este $sp:(p)(q)
  subu $sp,4
  sw $fp,0($sp)    # salvam $fp
  addiu $fp,$sp,0  # acum stiva este $sp:$fp:($fp v)(p)(q)
  lw $t0,4($fp)    # $t0 = p
  mul $t0,$t0,$t0  # $t0 = p*p
  subu $sp,4
  sw $t0,0($sp)    # acum stiva este $sp:(p*p)$fp:($fp v)(p)(q)
  lw $t0,8($fp)    # $t0 = q
  mul $t0,$t0,$t0  # $t0 = q*q
  lw $t1,0($sp)    # $t1 = p*p
  addu $sp,4       # acum stiva este $sp:$fp:($fp v)(p)(q)
  add $v0,$t1,$t0  # $v0 = p*p + q*q
  lw $fp,0($fp)    # restauram $fp
  addu $sp,4       # descarc cadrul de apel; acum stiva este iar $sp:(p)(q)
jr $ra
##########
```

Comentarii:
- functia returneaza valoarea in $v0; intrucat la revenirea in codul apelant conteaza noua valoare pusa de "f" in $v0, nici codul apelant nici functia nu trebuie sa salveze/restaureze acest registru;
- intrucat am avut parametri transmisi prin stiva, am folosit $fp pentru a-i accesa; aici el chiar a fost util, deoarece pe parcursul apelului stiva (si deci $sp) a mai fluctuat (prin incarcarea/descarcarea lui p*p).

Exemplu: returnarea prin memorie - anume prin parametri de iesire sau
~~~~~~~~ variabile statice asociate functiei special in acest scop;
tehnica nu este recomandata ca model de programare structurata, dar permite (si se foloseste la) returnarea unor valori de dimensiuni. mari (vectori, stringuri, etc.);
varianta cu parametri de iesirea fost deja descrisa in ex. anterioare si implementeaza de fapt un comportament de procedura (nu de functie) - trebuie sa dam noi explicit ca par. actual adresa locatiei unde se va ret. valoarea;
in urmatorul program ilustram ambele variante; practic, vom translata urmatorul program C:
~~~~~~~~

```
void f1(int *r, int p, int q){
  *r=p*p+q*q;
}

int f2r;  /* variabila statica asociata functiei f2 */

void f2(int p, int q){
  f2r=p*p+q*q;
}

int x,y;

void main(){
  f1(&x,1,2);        /* x devine 5 */
  f2(10,20); y=f2r;  /* y devine 500 */
}
```

Translatarea este:

```
.data
  x: .space 4
  y: .space 4
.text
main:
    # incarc parametrii actuali pentru f1
  subu $sp,12
  la $t0,x
  sw $t0,0($sp)
  li $t0,1
  sw $t0,4($sp)
  li $t0,2
  sw $t0,8($sp)  # acum stiva este $sp:(adr.x)(1)(2)
  jal f1
  addu $sp,12    # descarc parametrii actuali ai lui f1
    # acum x = 5
    # incarc parametrii actuali pentru f2
  subu $sp,8
  li $t0,10
  sw $t0,0($sp)
  li $t0,20
  sw $t0,4($sp)  # acum stiva este $sp:(10)(20)
  jal f2
  addu $sp,8     # descarc parametrii actuali ai lui f2
  lw $t0,f2r     # recuperez
  sw $t0,y       #   valoarea returnata
    # acum y = 500
li $v0,10
syscall
f1:              # la intrarea in apel stiva este $sp:(r)(p)(q)
  subu $sp,4
  sw $fp,0($sp)    # salvam $fp
  addiu $fp,$sp,0  # acum stiva este $sp:$fp:($fp v)(r)(p)(q)
  lw $t0,8($fp)    # $t0 = p
  mul $t0,$t0,$t0  # $t0 = p*p
  subu $sp,4
  sw $t0,0($sp)    # acum stiva este $sp:(p*p)$fp:($fp v)(r)(p)(q)
  lw $t0,12($fp)   # $t0 = q
  mul $t0,$t0,$t0  # $t0 = q*q
  lw $t1,0($sp)    # $t1 = p*p
  addu $sp,4       # acum stiva este $sp:$fp:($fp v)(r)(p)(q)
  add $t1,$t1,$t0  # $t1 = p*p + q*q
  lw $t0,4($fp)    # $t0 = r
  sw $t1,0($t0)    # *r = p*p + q*q
```

```
  lw $fp,0($fp)    # restauram $fp
  addu $sp,4       # descarc cadrul de apel;acum stiva este iar $sp:(r)(p)(q)
jr $ra
.data
  f2r: .space 4
    # variabila statica asociata functiei f2;
    # reamintim ca se aloca in continuarea lui y
.text
f2:              # la intrarea in apel stiva este $sp:(p)(q)
  subu $sp,4
  sw $fp,0($sp)    # salvam $fp
  addiu $fp,$sp,0  # acum stiva este $sp:$fp:($fp v)(p)(q)
  lw $t0,4($fp)    # $t0 = p
  mul $t0,$t0,$t0  # $t0 = p*p
  subu $sp,4
  sw $t0,0($sp)    # acum stiva este $sp:(p*p)$fp:($fp v)(p)(q)
  lw $t0,8($fp)    # $t0 = q
  mul $t0,$t0,$t0  # $t0 = q*q
  lw $t1,0($sp)    # $t1 = p*p
  addu $sp,4       # acum stiva este $sp:$fp:($fp v)(p)(q)
  add $t1,$t1,$t0  # $t1 = p*p + q*q
  la $t0,f2r       # $t0 = &f2r
  sw $t1,0($t0)    # f2r = p*p + q*q
  lw $fp,0($fp)    # restauram $fp
  addu $sp,4       # descarc cadrul de apel; acum stiva este iar $sp:(p)(q)
jr $ra
##########

Exemplu: returnarea prin stiva  - valoarea returnata este plasata in stiva
~~~~~~~~  acolo de unde codul apelant a inceput incarcarea datelor pt.
 apelul respectiv (suprascriind deci o parte din parametrii actuali),
 iar acolo se pune varful stivei; astfel codul apelant incarca parametrii,
 apeleaza functia, apoi descarca pur si simplu valoarea returnata;
  metoda este nestandard, se foloseste mai rar si este mai complicat de
 programat; translatam urmatorul program C:

int f(int p, int q){
  return p*p+q*q;
}

int x;

void main(){
  x=f(1,2);  /* x devine 5 */
}

Translatarea este:

.data
  x: .space 4
.text
main:
  subu $sp,8      # incarc parametrii actuali
  li $t0,1
  sw $t0,0($sp)
  li $t0,2
  sw $t0,4($sp)
    # acum stiva este $sp:(1)(2)
  jal f
    # acum stiva este $sp:(5)  (are in locul parametrilor valoarea returnata)
  lw $t0,0($sp)  # descarc valoarea
  addu $sp,4     #  returnata
  sw $t0,x       # si o recuperez (in x)
    # acum x = 5
```

```
li $v0,10
syscall
f:              # la intrarea in apel stiva este $sp:(p)(q)
  subu $sp,4
  sw $fp,0($sp)    # salvam $fp
  addiu $fp,$sp,0  # acum stiva este $sp:$fp:($fp v)(p)(q)
  lw $t0,4($fp)    # $t0 = p
  mul $t0,$t0,$t0  # $t0 = p*p
  subu $sp,4
  sw $t0,0($sp)    # acum stiva este $sp:(p*p)$fp:($fp v)(p)(q)
  lw $t0,8($fp)    # $t0 = q
  mul $t0,$t0,$t0  # $t0 = q*q
  lw $t1,0($sp)    # $t1 = p*p
  addu $sp,4       # acum stiva este $sp:$fp:($fp v)(p)(q)
  add $t1,$t1,$t0  # $t1 = p*p + q*q (rezultatul)
    # incepand de acum pregatesc stiva pentru retur
  sw $t1,8($fp)    # acum stiva este $sp:$fp:($fp v)(p)(rezultat)
  lw $fp,0($fp)    # restauram $fp; stiva este $sp:($fp v)(p)(rezultat)
  addu $sp,8       # descarc cadrul de apel si gloesc stiva pana la rezultat
    # acum stiva este $sp:(rezultat)
jr $ra
##########
```

Uneori valoarea returnata prin stiva este atat de mare incat suprascrie toti parametrii actuali si chiar cadrul de apel (care contine printre altele reg. callee-saved salvati si care trebuie restaurati); in acest caz intai deplasam spre stanga (spre adrese mici) parametrii actuali si cadrul de apel (sau macar partea de care avem nevoie mai tarziu, de ex. registrii callee-saved salvati), apoi plasam in stiva valoarea returnata, si apoi restauram registrii callee-saved salvati.

Exemplu: implementam o functie care primeste ca parametri doua worduri p si n
~~~~~~~~   si returneaza un vector ce contine de n ori wordul p;
          nu avem echivalent in limbajul C, deoarece in C o functie nu poate
           returna vectori (decat "imbracati" intr-un struct, dar si atunci
           dimensiunea lor este limitata superior);

```
.data
  s: .space 40  # buffer sufucient cat sa incapa un vector de 10 word-uri
.text
main:
    # incarc parametrii actuali p=7, n=8
  subu $sp,8
  li $t0,7
  sw $t0,0($sp)
  li $t0,8
  sw $t0,4($sp)
    # acum stiva este $sp:(7)(8)  (contine parametrii actuali)
  jal f
    # acum stiva este $sp:(7)(7)(7)(7)(7)(7)(7)(7)  (contine val. returnata)
    # descarc valoarea returnata si o recuperez in bufferul s
  la $t0,s
  li $s0,8
  et1:
    beqz $s0,et2
    lw $t1,0($sp)
    addu $sp,4
    sw $t1,0($t0)
    addu $t0,4
    subu $s0,1
    b et1
  et2:
    # acum de la adr.lui s s-au stocat 8 worduri egale cu 7, iar stiva e vida
li $v0,10
syscall
```
~~~~~~~~

```
f:  # la intrarea in apel stiva este $sp:(p)(n)
    # aloc un cadru de apel in care salvez $s0, $fp
  subu $sp,8
  sw $s0,0($sp)
  sw $fp,4($sp)
  addiu $fp,$sp,4
    # acum stiva este $sp:($s0 v)$fp:($fp v)(p)(n)
    # acum deplasez cadrul de apel si parametrii actuali(in total 4 word-uri)
    #   cu n-2 word-uri la stanga; translatez corespunzator si $sp, $fp
  lw $t0,8($fp)    # $t0 = n
  subu $t0,2       # $t0 = n-2
  sll $t0,$t0,2    # $t0 = 4*(n-2)  (distanta in octeti a deplasarii)
  subu $fp,$fp,$t0 # actualizam $fp
  move $t1,$sp     # indice sursa
  subu $sp,$sp,$t0 # actualizam $sp
  move $t0,$sp     # indice destinatie
  li $s0,4         # nr. de word-uri ce trebuie deplasate
  f_et1:
    lw $t2,0($t1)
    sw $t2,0($t0)
    addu $t1,4
    addu $t0,4
    subu $s0,1
    bgtz $s0,f_et1
   # acum stiva este $sp:($s0 v)$fp:($fp v)(p)(n)()()()()()()
   # construim valoarea returnata:
  lw $t0,8($fp)    # $t0 = n
  sll $t0,$t0,2    # $t0 = 4*n
  addu $t0,$fp,$t0 # stiva este $sp:($s0 v)$fp:($fp v)(p)(n)()()()()()$t0:()
  lw $s0,8($fp)    # $s0 = n
  lw $t1,4($fp)    # $t1 = p
  f_et2:
    sw $t1,0($t0)
    subu $t0,4
    subu $s0,1
    bgtz $s0,f_et2
   # acum stiva este $sp:($s0 v)$fp:$t0:($fp v)(7)(7)(7)(7)(7)(7)(7)(7)
   # restauram $s0, $fp
  lw $s0,-4($fp)
  lw $fp,0($fp)
  addu $sp,8
   # acum stiva este $sp:(7)(7)(7)(7)(7)(7)(7)(7)
jr $ra
##########

Comentarii:
- am deplasat la stanga cu n-2 si nu cu n word-uri deoarece dupa construirea
 valorii returnate in stiva ne mai trebuie doar $s0 vechi si $fp vechi, nu si
 p si n;
- puteam construi de la bun inceput cadrul de apel in pozitia translatata,
 dar asa respectam conventiile MIPS de a construi cadrul de apel la inceputul
 apelului; dealtfel, inainte de a construi valoarea returnata e posibil sa
 avem de efectuat si alte prelucrari (ocazie cu care stiva ar putea fluctua)
 si atunci e bine sa avem cadrul de apel in pozitia uzuala;
- dupa deplasare entitatile din cadrul de apel si parametrii actuali au
 aceleasi offset-uri (fixe) fata de $fp (anticipabile in faza dezvoltarii
 programului) ca si inaintea deplasarii; deci putem face prelucrari si dupa
 deplasarea cadrului de apel, iar majoritatea instructiunilor se scriu la fel
 ca in cazul in cadrul n-ar fi fost deplasat (si deci sunt usor de generat
 dintr-un compilator ce translateaza alt limbaj in MIPS);
- in faza elaborarii programului se poate anticipa dimensiunea blocului
 translatat (aici 4 word-uri) si adresa de unde incepe scrierea in stiva a
 valorii returnate (aici $fp+4*n sau $sp + 4*(n+1)).
```

CONVENTIE MIPS: Functiile vor returna valori doar prin $v0 si $v1.

In exemplele care urmeaza, din considerente didactice (pentru a ne obisnui cu stiva), ne luam libertatea de a returna valori si prin stiva daca vom considera ca este util/interesant.

1e) Conventiile Pascal/C:

Am spus ca exista mai multe reguli posibile in ceea ce priveste ordinea incarcarii parametrilor actuali in stiva, ordinea alocarii variabilelor locale automatice in stiva, modul de returnare a valorii sau cine descarca stiva la sfarsitul apelului (codul apelant sau rutina apelata). Dintre acestea importante sunt ordinea de incarcare a parametrilor actuali si cine descarca stiva la sfarsit - aceste reguli precizeaza modul in care comunica rutina apelata si codul apelant (ambele trebuie sa respecte aceleasi conventii pentru a putea comunica corect).
In practica se folosesc urmatoarele doua conventii:
- conventia Pascal: parametrii sunt push-ati in ordinea in care apar in declaratia procedurii (deci langa $fp este parametrul declarat cel mai la dreapta), iar stiva este descarcata de rutina apelata;
- conventia C: parametrii sunt push-ati in ordinea inversa celei in care apar in declaratia procedurii (deci langa $fp este parametrul declarat cel mai la stanga), iar stiva este descarcata de codul apelant; aceasta conventie a fost folosita in exemplele de pana acum.

In plus, functiile Pascal rezerva o locatie automatica pentru stocarea valorii returnate (pentru a putea propune de mai multe ori pe perioada apelului o valoare de returnat, fara sa se iasa din apel, prin instructiuni de forma nume_functie := expresie) - la revenire valoarea se ia de acolo (si de exempu se copiaza in $v0).
Urmatorul exemplu ilustreaza aceste conventii:

Exemplu: functie ce primeste ca parametri un word "a" prin referinta/adresa si un word "b" prin valoare si returneaza maximul dintre cele doua worduri;
~~~~~~~~
prezentam codul Pascal/C si cate un echivalent MIPS neoptimizat (deci nu asa cum il genereaza compilatoarele acestor limbaje):

varianta Pascal:

```
  function max(var a:int; b:int):int;
    begin
     if a>b then max:=a else max:=b
    end;
  var x,y:int;
  begin
  x:=10;
  y:=max(x,5)
  end.
```

echivalent MIPS:

```
.data
  x: .space 4
  y: .space 4
.text
main:
  li $t0,10
  sw $t0,x       # x=10
  subu $sp,8     # incarc parametrii in ordinea
  la $t0,x       #   in care apar in declaratia functiei:
  sw $t0,4($sp)  #   intai a, apoi b
  li $t0,5       # deci ultimul incarcat (si aflat in varful stivei)
  sw $t0,0($sp)  #   va fi b
```
~~~~~~~~

```
    # acum stiva este $sp:(b)(a)
  jal max
    # acum stiva este vida (stiva e descarcata de "max"),
    # iar val. ret. este in $v0
  sw $v0,y
li $v0,10
syscall
max:  # la intrarea in apel stiva este $sp:(b)(a)
    # rezerv loc pentru salvarea $fp si pentru valoarea de retur (folosita in
    #  instructiunile "max:=...")
  subu $sp,8
  sw $fp,4($sp)
  addiu $fp,$sp,4
    # acum stiva este $sp:(loc.val.ret)$fp:($fp v)(b)(a)
  lw $t0,8($fp)
  lw $t0,0($t0)  # $t0 = valoarea de la adresa a
  lw $t1,4($fp)  # $t1 = b
  bgt $t0,$t1,max_et1
    sw $t1,-4($fp)  # max := b
    b max_et2
  max_et1:
    sw $t0,-4($fp)  # max := val. de la adr. a
  max_et2:
  lw $v0,-4($fp)    # initializez registrul de retur $v0 cu val. ret.
  lw $fp,0($fp)     # restaurez $fp
  addu $sp,16       # descarc cadrul de apel si parametrii actuali
    # acum stiva este vida
jr $ra
##########

varianta C:

  int max(int *a, int b){
    if(*a>b) return *a; else return b;
  }
  int x,y;
  void main(){
    x=10;
    y=max(&x,5);
  }

 echivalent MIPS:

.data
  x: .space 4
  y: .space 4
.text
main:
  li $t0,10
  sw $t0,x       # x=10
  subu $sp,8     # incarc parametrii in ordinea
  li $t0,5       #   inversa celei din declaratia functiei:
  sw $t0,4($sp)  #   intai b, apoi a
  la $t0,x       # deci ultimul incarcat (si aflat in varful stivei)
  sw $t0,0($sp)  #   va fi a
    # acum stiva este $sp:(a)(b)
  jal max
    # acum stiva este $sp:(a)(b)
  addu $sp,8     # stiva e descarcata de codul apelant
  sw $v0,y       # val. ret. este in $v0
li $v0,10
syscall
max:  # la intrarea in apel stiva este $sp:(a)(b)
    # rezerv loc doar pentru salvarea $fp
```

```
  subu $sp,4
  sw $fp,0($sp)
  addiu $fp,$sp,0
    # mergea si "move $fp,$sp" dar "addiu $fp,$sp,0" e mai rapid (are
    #  corespondent direct in cod masina, in timp ce "move" e
    #  pseudoinstructiune)
    # acum stiva este $sp:$fp:($fp v)(a)(b)
  lw $t0,4($fp)
  lw $v0,0($t0)  # $v0 = *a
  lw $t0,8($fp)  # $t0 = b
  bgt $v0,$t0,max_et
    lw $v0,8($fp)
  max_et:
    # acum $v0 contine val. ret. (*a sau b)
  lw $fp,0($fp)     # restaurez $fp
  addu $sp,4        # descarc cadrul de apel
    # acum stiva este ca la intrarea in apel $sp:(a)(b)
jr $ra
##########

1f) Proceduri cu numar variabil de parametri:
---------------------------------------------

 In limbajul C (nu si in Pascal) exista functii cu numar variabil de
parametri; la declarare se precizeaza lista fixa si lista variabila de
parametri formali; lista fixa este neaparat in stanga, iar cea variabila
este desemnata de "..." si apare in dreapta.
 De exemplu un program cu o functie ce returneaza suma unui numar oarecare
de intregi este:

  #include<stdarg.h>
  int aduna(int n, ...){
    int i,s;
    va_list l;
    va_start(l,n);
    s=0;
    for(i=0;i<n;++i) s+=va_arg(l,int);
    va_end(l);
    return s;
  }
  int s;
  void main(){
    s=aduna(3,1,2,3);  /* 3=nr. numerelor, 1,2,3=numerele; obtinem s=6 */
    s=aduna(2,10,20);  /* obtinem s=30 */
  }

comentarii:
- "va_list", "va_start", "va_end", "va_arg" sunt descrise in "stdarg.h":
 "va_list" este un tip de date gen pointer (implementat a.i. sa poata pointa
    par. actuali din stiva); BC++ 3.1 il implementeaza ca pointer la void;
 "va_start(l,n)" e un macro ce face ca "l" sa pointeze dupa parametrul actual
    "n" (octetul urmator locatiei lui n din stiva);
 "va_arg(l,int)" avanseaza pointerul "l" pana dupa urmatorul parametru
    actual (presupunand ca acesta este de tip int), apoi returneaza valoarea
    parametrului peste care a trecut convertita la tipul indicat (int);
    practic, o invocare "va_arg(l,tip)" returneaza ceva de forma:

          *(tip *)  ( ((*(char **)&l) += sizeof(tip)) - sizeof(tip) )

 "va_end(l)" elibereaza resursele folosite de "l" (dupa "va_end(l)" nu mai
    pot face "va_arg(l,ceva)" decat daca incep iar cu "va_start");
- compilatorul de C nu implementeaza un mecanism de autodetectare in functie
 a listei de par. actuali incarcati - programatorul trebuie sa comunice
 functiei explicit informatiile necesare; aceasta se poate face de ex. prin
```

intermediul par. din lista fixa - noi am transmis nr. numerelor prin n;
daca in functie incercam sa procesam mai multi/mai putini parametri decat am dat, nu se genereaza eroare la compilare, dar la executie stiva va fi exploatata fie prea mult, fie prea putin (de ex. daca dupa "for" mai fac un "va_arg(l,int)" se iau urmatorii sizeof(int) octeti din stiva de dupa ultimul par. actual, se priveste informatia din ei ca "int" si se proceseaza); ceva asemanator se intampla si daca procesam par.actuali cu alt tip decat i-am incarcat (de ex. am incarcat valori "int" iar in functie ii procesez cu "va_arg(l,double)") - octetii din stiva se vor procesa grupati si interpretati altfel decat i-am incarcat; deci e sarcina programatorului sa apeleze functia cu exact atatia par. actuali cati proceseaza ea, iar in apel sa-i proceseze cu tipul corect;
- in ceea ce priveste codul MIPS/masina generat de compilator, faptul ca parametrii din lista fixa trebuie declarati intotdeauna primii face ca, in baza conventiei C, ei sa fie incarcati in stiva mereu ultimii (adica langa $fp) si astfel vor avea offset-uri constante fata de $fp, indiferent de nr. par. actuali incarcati in total - astfel este posibila generarea unui cod MIPS/masina unic, care sa functioneze corect indiferent de nr. par. actuali;
- daca compilatorul ar folosi conventia Pascal ca stiva sa fie descarcata de par. actuali in functie, ar fi imposibil sa deduca cu cat trebuie descarcata fara a consulta textul codului apelant (acolo e scris cati par. actuali se incarca) si astfel functia nu ar putea fi dezvoltata intr-o biblioteca independenta care sa fie ulterior linkeditata cu acest program (sau cu altele); intrucat in C codul care incarca stiva este si cel care o descarca, compilatorul gaseste in el toate informatiile necesare si astfel atat functia cat si restul programului pot fi dezvoltate/compilate separat si linkeditate impreuna;
acelasi motiv (anume ca din textul functiei nu rezulta cati par. actuali s-au incarcat) face imposibila implementarea returnarii valorii prin stiva - intr-adevar, val. returnata ar trebui plasata acolo de unde a inceput incarcarea par. actuali, iar compilatorul nu poate determina acest loc doar din textul (C al) functiei.

Exemplu: Prezentam un cod MIPS echivalent cu programul C de mai inainte:
~~~~~~~~

```
.data
  s: .space 4
.text
main:
  subu $sp,16   # incarcam parametrii, conform conventiei C
  li $t0,3
  sw $t0,12($sp)
  li $t0,2
  sw $t0,8($sp)
  li $t0,1
  sw $t0,4($sp)
  li $t0,3
  sw $t0,0($sp)
  jal aduna      # functia returneaza in $v0
  addu $sp,16    # descarcam atatia par. cati am incarcat (adica 4 parametri)
  sw $v0,s       # val. ret. este in $v0
    # acum s contine 6
  subu $sp,12
  li $t0,20
  sw $t0,8($sp)
  li $t0,10
  sw $t0,4($sp)
  li $t0,2
  sw $t0,0($sp)
  jal aduna
  addu $sp,12     # acum am incarcat doar 3 par., deci descarc doar 3 par.
  sw $v0,s
```
~~~~~~~~

```
    # acum s contine 30
li $v0,10
syscall
aduna:   # primeste o stiva de forma $sp:(n)()()()...
  subu $sp,16      # rezerv loc pt. salvat $fp si pentru i,s,l din functie
  sw $fp,12($sp)
  addiu $fp,$sp,12
    # acum stiva este $sp:(l)(s)(i)$fp:($fp v)(n)()()()...
  addu $t0,$fp,8
  sw $t0,-12($fp)  # va_start(l,n); avem $sp:(l)(s)(i)$fp:($fp v)(n)l:()()...
  sw $zero,-8($fp) # s=0
  sw $zero,-4($fp) # i=0
  aduna_et1:
  lw $t0,-4($fp)
  lw $t1,4($fp)
  bge $t0,$t1,aduna_et2  # daca i>=n iesim
    lw $t0,-12($fp)
    addu $t0,4           # 4 este sizeof(int)
    sw $t0,-12($fp)
    lw $t0,-4($t0)       # 4 este sizeof(int)
    lw $t1,-8($fp)
    add $t1,$t1,$t0
    sw $t1,-8($fp)       # s+=va_arg(i,int)
    lw $t0,-4($fp)
    add $t0,1
    sw $t0,-4($fp)       # ++i
    b aduna_et1
  aduna_et2:
    # in aceasta implementare nu avem ce executa pentru va_end(l)
  lw $v0,-8($fp)
  lw $fp,0($fp)
  addu $sp,16
jr $ra
##########

O implementare mai eficienta a functiei "aduna" dar care nu traduce la fel de
fidel programul C este urmatoarea:

aduna:   # primeste o stiva de forma $sp:(n)()()()...
  subu $sp,12      # salvez $fp,$s0,$s1
  sw $fp,8($sp)
  sw $s0,4($sp)
  sw $s1,0($sp)
  addiu $fp,$sp,8
   # stiva este $sp:($s1 v)($s0 v)$fp:($fp v)(n)()()()...
   # convin sa aloc s in $v0, i in $s0, l in $s1
  addu $s1,$fp,8    # va_start(l,n)
   # avem $sp:($s1 v)($s0 v)$fp:($fp v)(n)$s1:()()()...
  li $v0,0          # s=0
  li $s0,0          # i=0
  lw $t0,4($fp)     # $t0=n
  aduna_et1:
  bge $s0,$t0,aduna_et2
    addu $s1,4      # 4 este sizeof(int)
    lw $t1,-4($s1)
    add $v0,$v0,$t1 # s+=va_arg(i,int)
  addu $s0,1      # ++i
  b aduna_et1
  aduna_et2:
    # nici in aceasta implementare nu avem ce executa pentru va_end(l)
  lw $s0,-4($fp)    # restaurez $s0,$s1,$fp
  lw $s1,-8($fp)
  lw $fp,0($fp)
  addu $sp,12       # elimin cadrul de apel
```

```
jr $ra              # $v0 contine deja valoarea ce trebuie returnata
##########
```

Comentarii:
- obs. ca par. din lista fixa (adica "n") a fost accesat cu un deplasament constant (anume 4) fata de $fp, indiferent de nr. par. actuali incarcati;
- daca vrem sa scriem direct in MIPS programul cu functia cu nr. variabil de parametri putem implementa manual orice comportament - de exemplu consultand in functie par. "n" putem deduce cati par. actuali s-au incarcat si putem scrie codul a.i. sa descarcam stiva in functie (sau sa returnam valoarea prin stiva);
  un compilator de C nu poate sa faca asta automat pt. ca nu stie ca "n" va purta in el informatia referitoare la numarul par. actuali si e foarte greu sa deduca acest lucru din modul cum e folosit "n" in functie (de ex. sa vada ca se fac in total "n" invocari "va_arg") - asta tine de gandirea programatorului, compilatorul ar trebui sa analizeze algoritmul implementat pt. a o descoperi, ceea ce e f. greu, asa ca pt. el "n" este doar un par. ca toti ceilalti;
  modul de implementare manuala a comportamentului dorit va fi facut de programator in mod diferit de la o procedura/functie la alta, in functie de maniera de transmitere a informatiei privind nr. par. actuali (prin ce par. s-a transmis aceasta inf., etc.) - deci in fct. de algoritmul implementat in procedura/functie.

1g) Proceduri apelate din alte proceduri; proceduri recursive:

In exemplele de pana acum am salvat pe perioada unui apel vechea valoare a lui $fp desi nu a fost necesar - la revenirea din apel codul apelant nu a avut nevoie de vechea valoare a lui $fp; am vrut doar sa ilustram un mod general de lucru - $fp este callee-saved iar apelatul l-a folosit pentru a-si accesa entitatile din stiva. In schimb nu am salvat $ra, desi si el este callee-saved.

Salvarea/restaurarea lui $fp de catre apelat este necesara atunci cand apelantul este si el o procedura (care a folosit si el $fp pentru a-si accesa parametrii actuali si cadrul de apel iar dupa revenire va dori sa o poata face din nou). Salvarea/restaurarea lui $ra de catre apelat este necesara atunci cand si acesta apeleaza alte proceduri (deci este apelant pentru ele, iar apelarea modifica $ra) iar dupa revenirea din apelurile imbricate va avea nevoie de vechea valoare a lui $ra pentru a putea face si el revenirea in apelantul sau.

Avand in vedere conventiile MIPS si faptul ca $fp si $ra sunt callee-saved, deci trebuie salvati/restaurati de apelat (doar) daca acesta ii modifica, rezulta ca:
- o procedura va salva/restaura $fp daca primeste parametri prin stiva sau isi aloca un cadru de apel in stiva si atunci conform uzantelor foloseste $fp pentru a accesa aceste entitati;
- o procedura va salva/restaura $ra daca apeleaza alte proceduri.

Un caz cand apar ambele situatii este cel al procedurilor/functiilor recursive.

Notam ca folosirea stivei in implementarea apelurilor de proceduri/functii este indispensabila doar in cazul recursiei. Daca in limbajul nostru nu admitem recursii (directe sau indirecte) stiva nu este necesara, parametrii si cadrele de apel se pot aloca static - pentru fiecare procedura/functie rezervam in zona de date statice cate o zona ce va contine atat variabilele locale cat si parametrii (neavand recursii, nu apare necesitatea ca acestea sa aibe mai multe instante la un moment dat - cate una pentru fiecare apel inceput si inca neterminat al procedurii/functiei respective - deoarece in orice moment este activ cel mult un apel al ei). De exemplu versiunile vechi de Fortran nu admiteau recursivitatea, deoarece cadrele alocate static produceau pe unele masini mai vechi coduri mai rapide.

Urmatorul exemplu ilustreaza apelarea unei proceduri din alta procedura:

Exemplu: calculul celui de-al n-lea termen al sirului Fibonacci, folosind
~~~~~~~~ o functie care aduna doi termeni si o procedure care translateaza
          termenii; functia returneaza prin stiva;
 mai exact vom translata urmatorul cod C:

```
  int aduna(int a, int b){
    return a+b;
  }
  void iteratie(int *a, int *b){
    int c;
    c=aduna(*a, *b);
    (*a)=(*b); (*b)=c;
  }
  void main(){
    int n=5,x=1,y=1,z;
    register int i;
    for(i=2;i<n;++i)iteratie(&x,&y);
    z=y;
  }
```

Translatarea este:

```
.data
  n: .word 5
  x: .word 1
  y: .word 1
  z: .space 4
.text
main:
  li $s0,2 # alocam i in $s0
  inceput:
  lw $t0,n
  bge $s0,$t0,sfarsit
    subu $sp,8    # incarcam parametrii pt. "iteratie", conform conventiei C
    la $t0,y
    sw $t0,4($sp)
    la $t0,x
    sw $t0,0($sp) # $sp:(&x)(&y)
    jal iteratie
    addu $sp,8    # descarcam stiva de par. actuali ai lui "iteratie"
  add $s0,$s0,1
  b inceput
  sfarsit:
  lw $t0,y
  sw $t0,z  # acum z contine 5
li $v0,10
syscall
iteratie: # primeste parametri adrese $sp:(a)(b)
    # alocam un cadru de apel in care vom salva $ra, $fp, si vom aloca c
  subu $sp,12
  sw $ra,8($sp)
  sw $fp,4($sp)
  addiu $fp,$sp,8  # acum $sp:(c)($fp v)$fp:($ra v)(a)(b)
    # incarc parametrii actuali pentru "aduna"
  subu $sp,8
  lw $t0,8($fp)
  lw $t0,0($t0)
  sw $t0,4($sp)  # push *b
  lw $t0,4($fp)
  lw $t0,0($t0)
  sw $t0,0($sp)  # push *a
    # acum $sp:(*a=x)(*b=y)(c)($fp v)$fp:($ra v)(a=&x)(b=&y)
  jal aduna
    # am convenit ca "aduna" sa returneze prin stiva, deci acum:
```
~~~~~~~~

```
    # $sp:(x+y)(c)($fp v)$fp:($ra v)(a=&x)(b=&y)
  lw $t0,0($sp)
  addu $sp,4      # pop valoarea returnata de "aduna"
    # acum $sp:(c)($fp v)$fp:($ra v)(a=&x)(b=&y)
  sw $t0,-8($fp)  # c=aduna(*a,*b)
  lw $t0,8($fp)
  lw $t0,0($t0)
  lw $t1,4($fp)
  sw $t0,0($t1)   # (*a)=(*b), adica x=y
  lw $t0,-8($fp)
  lw $t1,8($fp)
  sw $t0,0($t1)   # (*b)=c, adica y=c
    # restauram registrii calee-saved, descarcam cadrul de apel si revenim
  lw $ra,0($fp)
  lw $fp,-4($fp)
  addu $sp,12
jr $ra
aduna: # primeste parametri intregi $sp:(a)(b)
    # alocam un cadru de apel in care salvam $fp
  subu $sp,4
  sw $fp,0($sp)
  addiu $fp,$sp,0
    # acum stiva (desenata vertical) este:
    #   $sp:$fp:($fp vechi salvat de "aduna" si care pointeaza |)
    #           (a al lui "aduna" = x)                         |
    #           (b al lui "aduna" = y)                         |
    #           (loc c al lui "iteratie")                      |
    #           ($fp vechi salvat de "iteratie")               |
    #           ($ra vechi salvat de "iteratie")  <------------|
    #           (a al lui "iteratie" = &x)
    #           (b al lui "iteratie" = &y)
  lw $t0,4($fp)
  lw $t1,8($fp)
  add $t0,$t0,$t1  # $t0=a+b
    # pregatim stiva cu valoarea returnata
  sw $t0,8($fp)
  lw $fp,0($fp)
  addu $sp,8
jr $ra
##########
```

Comentarii:
- "iteratie" a fost obligat sa salveze/restaureze $ra, deoarece "jal aduna" il modifica iar dupa revenirea din "aduna" procedura "iteratie" are nevoie de $ra-ul sau pentru a face revenirea in "main";
- "aduna" a fost obligata sa salveze/restaureze $fp, deoarece dupa revenire "iteratie" are nevoie de $fp-ul sau pentru a-si accesa variabila c (cu deplasamentul cunoscut de el, -8).

Observatie: exemplele din [1] par a sugera (desi nu este clar daca este vorba de o conventie MIPS) urmatoarea ordine de salvare a registrilor callee-saved: $sp:...($fp)($ra) iar daca vreunul din ei e absent, word-ul respectiv e lasat gol.

Prezentam si doua exemple de proceduri recursive:

Exemplu: problema turnurilor din Hanoi, folosind o procedura recursiva;
~~~~~~~~
* enuntul problemei:
  avem 3 pari (numerotati 1,2,3); pe parul 1 avem o stiva de n discuri, cu diametrele descrescatoare de la baza spre varf; putem face mutari - o mutare consta in mutarea discului de sus de pe un par pe altul, cu conditia sa nu-l punem peste un disc mai mic decat el; vrem o succesiune de mutari prin care tot teancul de discuri sa fie mutat de pe parul 1 pe parul 3;
~~~~~~~~

```
* solutie (de complexitate exponentiala):
  notam h(k,a,b,c) = succesiunea de mutari necesare mutarii unei stive de pe
 parul a pe parul b, folosind parul c ca auxiliar; atunci:
  daca k=1, h(1,a,b,c)=ab;
  daca k>1, h(k,a,b,c)=h(k-1,a,c,b) ab h(k-1,c,b,a)
 (intr-adevar, cand deplasam primele k-1 discuri il putem ignora pe cel de-al
 k-lea, care este cel mai mare si este jos de tot - deci putem pune orice
 peste el);
* implementare:
 mutarile succesive vor fi scrise intr-un vector "m" de word ca perechi de
  numere (1,2 sau 1,3 sau 2,3);
 vom folosi o procedura h(k,a,b,c) cu 4 parametri word (primul e nr. de
  discuri ce trebuie mutate, ultimii sunt parii folositi in ordinea descrisa
  mai sus); ea va lucra astfel:
    daca k=1, atunci scrie in "m" (de la pozitia curenta): a,b;
    altfel, apeleaza (recursiv) h(k-1,a,c,b), apoi scrie in "m" (de la
      pozitia curenta) a,b, apoi apeleaza recursiv h(k-1,c,b,a);
 pozitia curenta in "m" va fi urmarita global cu $s0;
 in programul principal vom apela h(n,1,3,2);
programul este:

.data
  m: .space 400
  n: .word 3
.text
main:
  la $s0,m     # $s0 va contine mereu adresa word-ului curent din m
  subu $sp,16  # incarcam parametrii pentru h(n,1,3,2), conform conventiei C
  li $t0,2
  sw $t0,12($sp)
  li $t0,3
  sw $t0,8($sp)
  li $t0,1
  sw $t0,4($sp)
  lw $t0,n
  sw $t0,0($sp)
  jal h         # h(n,1,3,2)
  addu $sp,16   # descarcam stiva de parametrii actuali
    # acum m contine 1,3,1,2,3,2,1,3,2,1,2,3,1,3
li $v0,10
syscall
h:
  subu $sp,8
  sw $ra,4($sp)
  sw $fp,0($sp)
  addiu $fp,$sp,4
    # $sp:($fp v)$fp:($ra v)(k)(a)(b)(c)...
  lw $t0,4($fp)
  li $t1,1
  beq $t0,$t1,et1
    # daca k>1 fac h(k-1,a,c,b)
    subu $sp,16
    lw $t0,12($fp)
    sw $t0,12($sp)
    lw $t0,16($fp)
    sw $t0,8($sp)
    lw $t0,8($fp)
    sw $t0,4($sp)
    lw $t0,4($fp)
    subu $t0,1
    sw $t0,0($sp)
    jal h
    addu $sp,16
  et1: # scriu a,b
```

```
    lw $t0,8($fp)
    sw $t0,0($s0)
    lw $t0,12($fp)
    sw $t0,4($s0)
    addu $s0,8
  lw $t0,4($fp)
  li $t1,1
  beq $t0,$t1,et2
    # daca k>1 fac h(k-1,c,b,a)
    subu $sp,16
    lw $t0,8($fp)
    sw $t0,12($sp)
    lw $t0,12($fp)
    sw $t0,8($sp)
    lw $t0,16($fp)
    sw $t0,4($sp)
    lw $t0,4($fp)
    subu $t0,1
    sw $t0,0($sp)
    jal h
    addu $sp,16
  et2:
  lw $ra,0($fp)
  lw $fp,-4($fp)
  addu $sp,8
jr $ra
##########
```

Comentarii:
- pentru a testa a doua oara daca k>1 (inainte de a face h(k-1,c,b,a)) a trebuit sa reincarcam $t0, $t1 cu aceleasi valori, deoarece i-am alterat de exemplu la scrierea lui a,b si oricum ei nu s-au pastrat in urma apelului h(k-1,a,c,b);
- evolutia stivei este (am notat ram = adresa de retur din program, adica $ra salvat la primul nivel de apel al lui h, rah = adresa de retur din procedura, adica $ra salvat la nivelele superioare de apel al lui h, fpi = diverse valori ale lui $fp salvate de apeluri (am desenat deasupra si unde pointeaza ele):

```
h(3,1,3,2):
|(fp0)(ram)(3)(1)(3)(2)
|h(2,1,2,3):                fp1
||                           V
||(fp1)(rah)(2)(1)(2)(3)(fp0)(ram)(3)(1)(3)(2)
||
||h(1,1,3,2):                fp2                   fp1
|||                           V                     V
|||(fp2)(rah)(1)(1)(3)(2)(fp1)(rah)(2)(1)(2)(3)(fp0)(ram)(3)(1)(3)(2)
|||
||| ==> se scrie: 13
|||_
||
||==> se scrie: 12          fp1
||                           V
||(fp1)(rah)(2)(1)(2)(3)(fp0)(ram)(3)(1)(3)(2)
||
||h(1,3,2,1):                fp2                   fp1
|||                           V                     V
|||(fp2)(rah)(1)(3)(2)(1)(fp1)(rah)(2)(1)(2)(3)(fp0)(ram)(3)(1)(3)(2)
|||
||| ==> se scrie: 32
|||_
||_
|
```

```
|==> se scrie: 13
|
|(fp0)(ram)(3)(1)(3)(2)
|
|h(2,2,3,1):                fp1
||                           V
||(fp1)(rah)(2)(2)(3)(1)(fp0)(ram)(3)(1)(3)(2)
||
||h(1,2,1,3):                fp2                   fp1
|||                           V                     V
|||(fp2)(rah)(1)(2)(1)(3)(fp1)(rah)(2)(2)(3)(1)(fp0)(ram)(3)(1)(3)(2)
|||
||| ==> se scrie: 21
|||_
||
||==> se scrie: 23          fp1
||                           V
||(fp1)(rah)(2)(2)(3)(1)(fp0)(ram)(3)(1)(3)(2)
||
||h(1,1,3,2):                fp2                   fp1
|||                           V                     V
|||(fp2)(rah)(1)(1)(3)(2)(fp1)(rah)(2)(2)(3)(1)(fp0)(ram)(3)(1)(3)(2)
|||
||| ==> se scrie: 13
|||_
||_
|_
```

Exercitiu: modificati programul de mai sus a.i. mutarile sa fie afisate pe consola, sub forma unor linii de forma "a -> b" (unde a,b sunt parii sursa, respectiv destinatie).

Exemplu: calculul lui n! folosind o functie recursiva (pe baza metodei: daca n<=1 atunci n!=1, altfel n!=n*(n-1)!, iar pt. (n-1)! se apeleaza recursiv aceeasi functie); parametrii si valoarea returnata se transmit prin stiva:

```
.data
  n: .word 3
  x: .space 4
.text
main:
  subu $sp,4 # incarcam in stiva parametrul actual
  lw $t0,n
  sw $t0,0($sp)
  jal fact
  lw $t0,0($sp)
  addu $sp,4 # am convenit ca fct. sa ret. prin stiva si descarcam val. ret.
  sw $t0,x   # acum x contine 6
li $v0,10
syscall
fact:
  subu $sp,8
  sw $ra,4($sp)
  sw $fp,0($sp)
  addiu $fp,$sp,4 # $sp:($fp v)$fp:($ra v)(n)...
  lw $t0,4($fp)
  li $t1,1
  bgt $t0,$t1,et  # test n>1
    sw $t1,4($fp) # daca n<=1 plasez val. ret. 1 (din $t1) in stiva si inchei
    b sf
  et:
    subu $t0,1    # $t0 inca contine n
    subu $sp,4
```

```
    sw $t0,0($sp)
      # $sp:(n-1)($fp v)$fp:($ra v)(n)...
    jal fact
      # $sp:((n-1)!)($fp v)$fp:($ra v)(n)...
    lw $t0,0($sp)
    addu $sp,4      # $t0=(n-1)!, $sp:($fp v)$fp:($ra v)(n)...
    lw $t1,4($fp)   # $t1=n
    mul $t1,$t1,$t0 # $t1=n!
    sw $t1,4($fp)   # plasez val. ret. in stiva
  sf: # indiferent cum ajung aici avem $sp:($fp v)$fp:($ra v)(n!)...
  lw $ra,0($fp)
  lw $fp,-4($fp)
  addu $sp,8
jr $ra
##########
```

Comentariu: la adancimea maxima, stiva ajunge (ca mai inainte, am notat ram = adresa de retur din program, adica $ra salvat la primul nivel de apel al lui fact, raf = adresa de retur din functie, adica $ra salvat la nivelele superioare de apel al lui fact, fpi = diverse valori ale lui $fp salvate de apeluri (am desenat deasupra si unde pointeaza ele):

```
                      fp2          fp1
                       V            V
 $sp:(fp2)(raf)(1)(fp1)(raf)(2)(fp0)(ram)(3)
```

Notam ca daca intram de multe ori in apeluri fara sa mai si iesim sau daca incarcam parametri multi si/sau mari, la un moment dat spatiul stiva se va umple iar stiva va invada alte zone (de cod, date, etc.) ceea ce se soldeaza de obicei cu blocarea sau terminarea anormala a programului (de exemplu o recursie infinita va produce la un moment dat stack overflow).

1h) Metoda salturilor indirecte:

Este o metoda de a face salturi (de tip "goto") folosind mecanisme de revenire din proceduri (de tip "return"); in MIPS nu exista instructiuni specifice procedurilor (comportamentul de procedura este implementat explicit de programator folosind instructiunile uzuale) si de aceea aceasta metoda nu reprezinta un mecanism nou fata de cel care foloseste instructiunile de ramificare si salt (prezentate in lectia 2); prezentarea formei adaptate la MIPS a acestei metode este insa interesanta, deoarece ofera un exemplu de implementare a structurii "switch" din limbajul C si de cum putem apela proceduri pe baza unui indice selector.

Concret, este vorba de implementarea unor cicluri "while" ce itereaza structuri "switch" (in limbajul C) in care pe fiecare ramura trebuie apelata o procedura.
Mai exact, sa presupunem ca vrem sa implementam ceva de forma:

```
  while(c=get_comanda())
    switch(c){
      case valoare0: p0(); break;
      case valoare1: p1(); break;
      ...
    }
```

O modalitate de implementare MIPS ar fi de forma:

```
  et:
  jal get_comanda   # presupunem ca returneaza in $v0
   li $t0,valoare0
   bne $v0,$t0,et0
   jal p0
```

```
   b et
  et0:
   li $t0,valoare1
   bne $v0,$t0,et1
   jal p1
   b et
  et1:
   .....
  etn:   # cazul "default"
  b et
```

Aceasta modalitate are dezavantajul ca poate face multe comparatii inainte de a decide care procedura trebuie apelata. Putem elimina compararile succesive daca cream un "tabel de salt" "t" care pe pozitia i contine adresa procedurii "pi", "get_comanda" sa returneze in loc de "valoare0", "valoare1", ..., numere succesive 0, 1, ..., iar ciclul de mai sus sa se reduca la:

```
  et:
  jal get_comanda   # presupunem ca returneaza in $v0
  move $t0,$v0
  sll $t0,$t0,2     # acum $t0 contine offset-ul comp. de indice $v0 din t
                    #   fata de inceputul vectorului (vector de worduri)
  add $t0,t         # acum $t0 contine adr.de mem. a comp.de indice $v0 din t
  jalr $ra,$t0
  b et
```

O solutie care presupune tot un tabel de salt dar care in plus evita instr. "jal", "jalr" si "b" realizeaza transferul executiei de la o adresa la alta cu ajutorul unor instructiuni de tip "jr"(cu care se implementeaza mecanismul de "return") si s.n. metoda salturilor indirecte. Ea consta in efectuarea urmatorilor pasi:

1. incarca in $ra adresa "get_comanda" (procedura care presupunem ca furnizeaza in $v0 o valoare de retur dintr-un interval 0...n)
2. incarca intr-un registru, de ex. $t0, adresa din tabelul de salt "t" continuta in componenta de indice $v0 (in exemplul de mai sus este adresa procedurii "pi")
3. jr $t0

Instructiunea "jr $t0" de la pasul 3 transfera executia la procedura "pi"; la incheierea executarii acestei proceduri, "jr $ra"-ul ei va provoca reexecutarea lui "get_comanda" si reluarea de la pasul 1. Practic, codul va fi de forma:

```
  get_comanda:
   ... # cod ce pune in $v0 o valoare 0,...,n
  la $ra,get_comanda
  move $t0,$v0
  sll $t0,$t0,2
  add $t0,t
  jr $t0
```

(in codul de mai sus "get_comanda" nici nu mai este o procedura propriuzisa ci doar o eticheta de la care incepe un cod care calculeaza o valoare in $v0 si apoi efectueaza pasii 1-3 descrisi mai inainte).

Metoda salturilor indirecte este folosita in programele de tip interpretor (Basic-Sinclair, GW-Basic, etc.).

Exemplu: Emularea unui interpretor de comenzi de tip "command.com" foarte restrans: accepta de la intrare doar comenzi de un caracter, in fiecare caz efectul fiind scrierea la iesire a cate unui alt caracter; comenzile suportate si caracterul scris la exec. fiecareia din ele sunt:

- comanda 'a' --> caracterul '+' (adunare)
- comanda 's' --> caracterul '-' (scadere)
- comanda 'n' --> caracterul '!' (negare)

```
 pe langa aceste comenzi mai accepta si:
  comanda 't' --> produce terminarea programului
 daca la intrare vine alt caracter decat 'a', 's', 'n', se va scrie 'e'
 (eroare) - acest lucru se va realiza cu o comanda suplimentara (care se va
 executa automat in asemenea cazuri);
  fluxul de intrare va fi simulat printr-un string ce contine o succesiune de
 caractere (nume de comenzi sau alte caractere, care vor genera erori);
  fluxul de iesire va fi simluat printr-un string ce va contine caracterele
 scrise la executarea fiecarei comenzi;
  de exemplu daca stringul de intrare contine: "assaxnyyt"
         atunci stringul de iesire va contine: "+--+e!ee"

 programul este urmatorul:

.data
  comenzi: .ascii "asnte"
   # vector de byte cu numele comenzilor (doar primele 4 sunt accesibile
   #  utilizatorului);
   # pozitia in vector da codul de identificare 0...4 al comenzii
  t: .word aduna, scade, neaga, termina, eroare
   # tabel de salt (pe pozitia i (0...4) este adresa procedurii ce realizeaza
   #  comanda cu codul i)
  intrare: .ascii "assaxnyyt" # fluxul de intrare
  iesire: .space 100          # fluxul de iesire
.text
main:
  la $s0,intrare # cu $s0 parcurg sirul "intrare"
  la $s1,iesire  # cu $s1 parcurg sirul "iesire"
  get_comanda:
    lb $t0,0($s0)   # $t0=caracterul curent din fluxul de intrare
    addu $s0,1
    la $t1,comenzi  # $t1 va parcurge stringul "comenzi" de la poz. lui "a"
    addiu $t2,$t1,4 #  la cea a lui "e" (stocata in $t2)
   et1:
    lb $t3,0($t1)   # caut $t0 in "comenzi" (pana il gasesc sau pana $t1
    beq $t0,$t3,et2 #  indica comanda de eroare)
    addu $t1,1      #
    bne $t1,$t2,et1 #
   et2:
   sub $t1,$t1,$t2
   addu $t1,4       # acum $t1 este poz. (idf. numeric al) comenzii (4=err)
   sll $t1,$t1,2
   la $t2,t
   addu $t2,$t2,$t1
   lw $t2,0($t2)    # acum $t2 este adr. de mem. a proc. corespunzatoare
   la $ra,get_comanda
   jr $t2
##### proceduri de implementare a comenzilor: #####
aduna:          # la intrare $ra contine adr. lui "get_comanda"
  li $t0,'+'
  sb $t0,0($s1)
  addu $s1,1
  jr $ra        # transfera executia la "get_comanda"
scade:
  li $t0,'-'
  sb $t0,0($s1)
  addu $s1,1
  jr $ra
neaga:
  li $t0,'!'
  sb $t0,0($s1)
  addu $s1,1
  jr $ra
termina:
```

```
   # adaos penrtu a afisa sirul "iesire"
  sb $zero,0($s1)
  li $v0,4
  la $a0,iesire
  syscall
   # terminarea programului
  li $v0,10
  syscall
eroare:
  li $t0,'e'
  sb $t0,0($s1)
  addu $s1,1
  jr $ra
#####
```

Comentarii:
- vectorul de byte "comenzi" este ".ascii" si nu ".asciiz" deoarece logica programului garanteaza ca intotdeauna ne vom opri cel tarziu la a 5-a componenta, deci nu e nevoie sa marcam sfarsitul cu un terminator nul; vectorul de byte "intrare" este tot ".ascii" (si nu ".asciiz") deoarece logica programului face ca parcurgerea acestuia sa se termine la intalnirea lui "t", deci iarasi n-avem nevoie de terminatorul nul;
- am parcurs sirurile "intrare", "iesire" cu doi registri $s ($s0, resp. $s1) deoarece ei trebuie sa-si conserve valoarea curenta peste apelurile declansate cu "", iar acest lucru este asigurat de faptul ca registrii $s sunt callee-saved;
- observam ca programul apeleaza proceduri si revine din ele fara sa foloseasca instructiuni de tip "jal" (adica apel), ci doar "jr" (adica return);
- desi scopul comenzii "termina" este doar terminarea programului, in implementarea ei am adaugat si o parte pentru afisarea pe consola a sirului "iesire" rezultat, deoarece in fereastra "Data Segment" a lui PCSpim vedem doar codurile hexa ale caracterelor;
- putem realiza in acelasi mod un interpretor care sa execute comenzi avand nume mai lungi de un caracter si avand parametri (vezi exercitiile de la sfarsit).

Un alt exemplu de folosire a metodei salturilor indirecte ar putea fi un joc in care iterativ se citeste cate o tasta directionala si in functie de ea se deplaseaza pe ecran (cu ajutorul unor proceduri de desenare) un personaj in sus, jos, dreapta, stanga; in PCSpim este insa greu deoarece consola virtuala nu poate fi gestionata asa de flexibil; eventual vom simula miscarea pe ecran prin (re)pozitionarea unui caracter-personaj intr-o matrice de caractere-pozitii si afisarea de fiecare data a intregii matrici, iar tasta directionala va fi inlocuita cu o tasta obisnuita citita cu syscall (tema !).

1i) Corutine: - TODO

1j) Conventia MIPS a apelului de procedura:

In aceasta sectiune sintetizam toate conventiile MIPS legate de apelarea procedurilor pe care le-am descris pana acum - ele formeaza asa-numita conventie MIPS a apelului de procedura si sunt descrise in cartea [1], anexa A:

Apelantul, inainte de a face apelul imbricat, efectueaza urmatoarele:
1 - incarca parametrii pentru apelat; primii patru parametri se incarca in registrii $a0 - $a3 iar urmatorii in stiva (ei vor aparea imediat deasupra cadrului de apel al apelatului);
2 - salveaza registrii caller-saved ($a0 - $a3, $t0 - $t9) pe care doreste sa-i conserve peste apelul imbricat (adica la revenirea din apelul imbricat

 sa regaseasca aceleasi valori in ei);
3 - executa apelul (de ex. cu "jal").

Apelatul, la intrarea in apel, efectueaza urmatoarele:
1 - aloca cadrul de apel in stiva, scazand din $sp dimensiunea cadrului
 (care e constanta si este cunoscuta in faza de elaborare a programului,
 deci se poate specifica in cod printr-o valoare imediata); cadrul de apel va
 contine atat registrii callee-saved salvati de apelat cat si variabilele
 sale locale automatice (la adrese mari reg. callee-saved, la adrese mici
 variabilele locale automatice); cadrul trebuie sa aibe cel putin 24 octeti;
2 - salveaza (in cadrul de apel) registrii callee-saved ($s0 - $s7, $fp, $ra)
 pe care ii modifica; exemplele din cartea [1] sugereaza si o anumita ordine
 a lor in cadrul de apel; in general $fp trebuie salvat mereu (daca vrem sa
 accesam cu el parametrii primiti si informatiile din cadrul de apel) iar $ra
 doar daca din apelat facem apeluri imbricate;
3 - plaseaza $fp pe primul word din cadrul de apel, adunand la $sp
 "dimensiunea cadrului de apel minus patru" (valoare cunoscuta un faza de
 elaborare a programului).

Apelatul, la iesirea din apel, se asigura ca $sp este in acelasi loc ca dupa
pasul 3 de mai sus (deci s-au eliminat din stiva toate valorile temporare
incarcate de-a lungul apelului in continuarea cadrului de apel) apoi
efectueaza urmatoarele:
1 - daca e functie, pune valoarea ce trebuie returnata in $v0 si eventual
 $v1;
2 - restaureaza registrii callee-saved pe care i-a salvat la intrare;
3 - elimina cadrul de apel adunand la $sp dimensiunea cadrului (cunoscuta in
 faza de elaborare a programului);
4 - revine in apelant (executand de ex. "jr $ra").

 In exemplele care urmeaza vom respecta toate aceste conventii, cu
urmatoarele exceptii:
- toti parametrii vor fi transmisi prin stiva;
- in apelant vom salva registrii caller-saved INAINTE de incarcarea
 parametrilor pentru apelat; intr-adevar, daca am respecta ordinea pasilor
 din cartea [1], intre parametrii primiti de apelat prin stiva si cadrul de
 apel pe care acesta il aloca s-ar interpune diverse valori salvate de
 apelant, astfel incat offset-urile parametrilor fata de $fp-ul apelatului
 vor fi dificil de stabilit, mai ales daca apelul este facut de mai multe ori
 in contexte diferite in care numarul valorilor care se interpun variaza; in
 plus, $a0 - $a3 sunt caller-saved iar daca apelantul doreste sa conserve
 valorile lor peste apelul imbricat trebuie sa-i salveze inainte de a-i
 folosi pentru a transmite parametri apelatului;
- cadrul de apel nu va avea cel putin 24 octeti ci strict cat e nevoie;
- uneori valoarea returnata de functii va fi transmisa prin stiva.

 Exista situatii (mai rare) cand cadrul de apel are dimensiune variabila
(deci nu poate fi anticipata in faza de elaborare a programului si scrie in
cod printr-o valoare imediata) - de exemplu cand procedura are o variabila
locala automatica care este vector de o lungime data ca parametru (situatie
permisa si deci tratata de anumite compilatoare de C, ca de ex. "gcc").

Exemplu: program care afisaza valorile distincte dintr-un vector de intregi:
~~~~~~~~

Translatam urmatorul program C:

```
  #include<stdio.h>
  void distinct(int *v, int n){
    int d[n]; /* var. loc. auto, se aloca pe stiva de dim. n (dat ca par.) */
    register int i,j,k;
    k=0;
    for(i=0;i<n;++i){
      for(j=0;j<k;++j) if(v[i]==d[j]) break;
```
~~~~~~~~

```
      if(j==k) {d[k]=v[i]; ++k;}
    }
    for(i=0;i<k;++i) printf("%d ",d[i]);
    printf("\n");
  }

  int w[]={1,2,5,2,5,1,2,4};

  void main(){
    distinct(w,8); /* afisaza: 1 2 5 4 */
  }
```

Translatarea este:

```
.data
  w: .word 1,2,5,2,5,1,2,4
.text
main:
  subu $sp,8
  li $t0,8
  sw $t0,4($sp)
  la $t0,w
  sw $t0,0($sp)  # $sp:(adr.w)(8)
  jal distinct
  addu $sp,8
li $v0,10
syscall
distinct: # primeste: $sp:(v)(n)
    # calculam dimensiunea cadrului de apel
    # pt. asta, accesam par. cu $sp si nu folosim registri callee-saved
  lw $t0,4($sp) # $t0=n
  sll $t0,$t0,2 # $t0=4*n
  addu $t0,20   # $t0=4*n+20 (dim. cadrului de apel)
    # construim cadrul si salvam in el si dimensiunea lui
  sw $t0,-20($sp)   # (dim.cadru)()()()()$sp:(v)(n)
  sw $fp,-4($sp)    # (dim.cadru)()()()($fp v)$sp:(v)(n)
  subu $fp,$sp,4    # (dim.cadru)()()()$fp:($fp v)$sp:(v)(n)
  subu $sp,$sp,$t0  # $sp:()...()(dim.cadru)()()()$fp:($fp v)(v)(n)
  sw $s0,-4($fp)
  sw $s1,-8($fp)
  sw $s2,-12($fp)
    # restul cadrului de apel este d[0] ... d[n-1], deci avem:
    # $sp:(d[0])...(d[n-1])(dim.cadru)($s2 v)($s1 v)($s0 v)$fp:($fp v)(v)(n)
    # convenim sa alocam i in $s0, j in $s1, k in $s2
  li $s2,0      # k=0
  li $s0,0      # i=0
  distinct_ciclu1_inceput:
  lw $t0,8($fp) # $t0=n
  bge $s0,$t0,distinct_ciclu1_sfarsit
    li $s1,0      # j=0
    distinct_ciclu2_inceput:
    bge $s1,$s2,distinct_ciclu2_sfarsit
      lw $t0,4($fp)    # $t0=v
      move $t2,$s0     # $t2=i
      sll $t2,$t2,2    # $t2=i*4
      add $t0,$t0,$t2  # $t0=(unsigned char *)v+i*4=v+i
      lw $t0,0($t0)    # $t0=*(v+i)=v[i]
      lw $t1,-16($fp)  # $t1=dim.cadru
      subu $t1,4
      subu $t1,$fp,$t1 # $t1=d
      move $t2,$s1     # $t2=j
      sll $t2,$t2,2    # $t2=j*4
      add $t1,$t1,$t2  # $t1=(unsigned char *)d+j*4=d+j
      lw $t1,0($t1)    # $t1=*(d+j)=d[j]
```

```
      beq $t0,$t1,distinct_ciclu2_sfarsit # if(v[i]==d[j]) break;
    add $s1,1     # ++j
    b distinct_ciclu2_inceput
    distinct_ciclu2_sfarsit:
    beq $s1,$s2,distinct_et1 # if(j==k)...
    b distinct_et2
    distinct_et1:            # ...then
      lw $t0,4($fp)    # $t0=v
      move $t2,$s0     # $t2=i
      sll $t2,$t2,2    # $t2=i*4
      add $t0,$t0,$t2  # $t0=(unsigned char *)v+i*4=v+i
      lw $t0,0($t0)    # $t0=*(v+i)=v[i]
      lw $t1,-16($fp)  # $t1=dim.cadru
      subu $t1,4
      subu $t1,$fp,$t1 # $t1=d
      move $t2,$s2     # $t2=k
      sll $t2,$t2,2    # $t2=k*4
      add $t1,$t1,$t2  # $t1=(unsigned char *)d+k*4=d+k
      sw $t0,0($t1)    # *(d+k)=v[i], adica d[k]=v[i];
      add $s2,1        # ++k
    distinct_et2:
  add $s0,1     # ++i
  b distinct_ciclu1_inceput
  distinct_ciclu1_sfarsit:
    # afisare
  li $s0,0      # i=0
  distinct_ciclu3_inceput:
  bge $s0,$s2,distinct_ciclu3_sfarsit
    li $v0,1  # print int
    lw $t0,-16($fp)  # $t0=dim.cadru
    subu $t0,4
    subu $t0,$fp,$t0 # $t0=d
    move $t2,$s0     # $t2=i
    sll $t2,$t2,2    # $t2=i*4
    add $t0,$t0,$t2  # $t1=(unsigned char *)d+i*4=d+i
    lw $a0,0($t0)    # $t0=*(d+i)=d[i]
    syscall
    li $v0,4  # print string
    la $a0,distinct_blank
    syscall
  add $s0,1     # ++i
  b distinct_ciclu3_inceput
  distinct_ciclu3_sfarsit:
    li $v0,4  # print string
    la $a0,distinct_newline
    syscall
    # iesirea din apel
  lw $s0,-4($fp)   # restauram $s0
  lw $s1,-8($fp)   # restauram $s1
  lw $s2,-12($fp)  # restauram $s2
  lw $t0,-16($fp)  # $t0=dim.cadru
  lw $fp,0($fp)    # restauram $fp
  addu $sp,$sp,$t0 # eliminam cadrul de apel
jr $ra
.data
  distinct_blank: .asciiz " "
  distinct_newline: .asciiz "\n"
##########
```

Constatam ca in implementarea de mai sus fiecare accesare a lui "d" necesita un calcul prealabil al adresei de inceput a acestuia; in acest scop a trebuit sa salvam in stiva dimensiunea cadrului de apel (aceasta ne-a folosit si la sfarsit, ca sa eliminam cadrul de apel).

Putem lucra mai eficient daca salvam in stiva direct adresa de inceput a

lui "d"; mai general, daca avem de alocat in cadrul de apel mai multe entitati automatice de dimensiune variabila (asemeni lui "d") vom calcula o singura data la inceputul apelului adresele lor si le vom stoca in cadru imediat sub entitatile de dimensiune constanta - ele vor putea fi gasite folosind offset-uri constante fata de $fp, cunoscute in faza de elaborare a programului; dimensiunea cadrului de apel nu este necesar sa o salvam in stiva, deoarece daca stim ca $fp pointeaza primul word din cadru putem elimina cadrul atribuind lui $sp valoarea $fp+4.
 Cu aceste modificari procedura devine:

```
distinct: # primeste: $sp:(v)(n)
    # construim partea de dimensiune fixa a cadrului de apel
    #  continand $fp, $s0, $s1, $s2 salvati si locatia de salvare a adr. "d"
  subu $sp,20
  sw $fp,16($sp)
  sw $s0,12($sp)
  sw $s1,8($sp)
  sw $s2,4($sp)
  addiu $fp,$sp,16
    # acum $sp:()($s2 v)($s1 v)($s0 v)$fp($fp v)(v)(n)
    # calculam adr. de inceput a lui "d" si o salvam in cadru la -16($fp),
    #  apoi alocam "d" mutand $sp la adr. respectiva
  lw $t0,8($fp)    # $t0=n
  sll $t0,$t0,2    # $t0=4*n
  addiu $t0,$t0,16 # $t0=4*n+16
  subu $t0,$fp,$t0 # $t0=$fp-4*n-16=adr.lui "d"
  sw $t0,-16($fp)  # salvam adr. lui "d" in cadru
  move $sp,$t0     # alocam "d"
    # acum $sp:(d[0])...(d[n-1])(adr.d)($s2 v)($s1 v)($s0 v)$fp($fp v)(v)(n)
    # ca inainte, convenim sa alocam i in $s0, j in $s1, k in $s2
  li $s2,0      # k=0
  li $s0,0      # i=0
  distinct_ciclu1_inceput:
  lw $t0,8($fp) # $t0=n
  bge $s0,$t0,distinct_ciclu1_sfarsit
    li $s1,0      # j=0
    distinct_ciclu2_inceput:
    bge $s1,$s2,distinct_ciclu2_sfarsit
      lw $t0,4($fp)    # $t0=v
      move $t2,$s0     # $t2=i
      sll $t2,$t2,2    # $t2=i*4
      add $t0,$t0,$t2  # $t0=(unsigned char *)v+i*4=v+i
      lw $t0,0($t0)    # $t0=*(v+i)=v[i]
      lw $t1,-16($fp)  # $t1=d
      move $t2,$s1     # $t2=j
      sll $t2,$t2,2    # $t2=j*4
      add $t1,$t1,$t2  # $t1=(unsigned char *)d+j*4=d+j
      lw $t1,0($t1)    # $t1=*(d+j)=d[j]
      beq $t0,$t1,distinct_ciclu2_sfarsit # if(v[i]==d[j]) break;
    add $s1,1     # ++j
    b distinct_ciclu2_inceput
    distinct_ciclu2_sfarsit:
    beq $s1,$s2,distinct_et1 # if(j==k)...
    b distinct_et2
    distinct_et1:            # ...then
      lw $t0,4($fp)    # $t0=v
      move $t2,$s0     # $t2=i
      sll $t2,$t2,2    # $t2=i*4
      add $t0,$t0,$t2  # $t0=(unsigned char *)v+i*4=v+i
      lw $t0,0($t0)    # $t0=*(v+i)=v[i]
      lw $t1,-16($fp)  # $t1=d
      move $t2,$s2     # $t2=k
      sll $t2,$t2,2    # $t2=k*4
      add $t1,$t1,$t2  # $t1=(unsigned char *)d+k*4=d+k
```

```
      sw $t0,0($t1)    # *(d+k)=v[i], adica d[k]=v[i];
      add $s2,1        # ++k
    distinct_et2:
  add $s0,1     # ++i
  b distinct_ciclu1_inceput
  distinct_ciclu1_sfarsit:
    # afisare
  li $s0,0      # i=0
  distinct_ciclu3_inceput:
  bge $s0,$s2,distinct_ciclu3_sfarsit
    li $v0,1  # print int
    lw $t0,-16($fp)  # $t0=d
    move $t2,$s0     # $t2=i
    sll $t2,$t2,2    # $t2=i*4
    add $t0,$t0,$t2  # $t1=(unsigned char *)d+i*4=d+i
    lw $a0,0($t0)    # $t0=*(d+i)=d[i]
    syscall
    li $v0,4  # print string
    la $a0,distinct_blank
    syscall
  add $s0,1     # ++i
  b distinct_ciclu3_inceput
  distinct_ciclu3_sfarsit:
    li $v0,4  # print string
    la $a0,distinct_newline
    syscall
    # iesirea din apel
  lw $s0,-4($fp)   # restauram $s0
  lw $s1,-8($fp)   # restauram $s1
  lw $s2,-12($fp)  # restauram $s2
  addiu $sp,$fp,4  # plasam $sp (practic eliminam cadrul)
  lw $fp,0($fp)    # restauram $fp ($fp vechi inca se poate accesa cu 0($fp))
jr $ra
.data
  distinct_blank: .asciiz " "
  distinct_newline: .asciiz "\n"
##########
```

Unele exemple din cartea [1] sugereaza (fara a fi clar daca este o conventie MIPS) faptul ca programul principal poate fi privit tot ca o procedura "main" (apelata din kernel cu "jal main") si astfel el trebuie sa respecte conventiile scrise mai sus: sa aloce un cadru de apel (pe care sa-l dezaloce la sfarsit),sa salveze/restaureze registrii callee-saved pe care ii modifica, sa foloseasca $fp pentru a accesa informatiile din cadrul sau de apel, sa se termine cu "jr $ra" (revenind astfel la kernel), nu cu "li $v0,10" si "syscall" - aceasta varianta fiind folosita doar pentru terminari fortate (in genul functiei "abort" din limbajul C).
Urmatorul exemplu ilustreaza acest stil de lucru (pe care insa nu il vom folosi in celelalte exemple):

Exemplu: citirea unui vector de intregi, intregii fiind introdusi de la
~~~~~~~~   consola in format hexa. Functiile vor returna prin registrii.

Translatam urmatorul program C:

```
#include<stdio.h>

int getinthexa(){
  char buf[11];      /* vector local automatic de dim. fixa */
  register int i,n;  /* variabile locale register */
  scanf("%s",buf);   /* sau: fflush(stdin); gets(buf); */
  n=0;
  for(i=2;i<10;++i){
    n=n*16+buf[i]-'0';
```
~~~~~~~~

```
    if(buf[i]>='a')n=n-39; /* 39=('a'-'0')-10 */
  }
  return n;
}

int v[10];           /* vector global */

int main(){
  int n;             /* variabila locala automatica */
  register int i;    /* variabila locala register */
  scanf("%d",&n);
  for(i=0;i<n;++i){
    v[i]=getinthexa();
  }
  for(i=0;i<n;++i) printf("%d ",v[i]);
  printf("\n");
  return 0;
}

/* daca tastam:
     2
     0x0000001a
     0x00000102
   atunci se va afisa:
     26 258
*/

Translatarea este:

.data
  v: .space 40
  blank: .asciiz " "
  nl: .asciiz "\n"
.text
main:
    # "main" e o functie ce necesita un cadru de apel
    #   in care alocam "n" si salvam $s0 (va fi "i"), $fp;
    # functia "main" returneaza 0 prin $v0
  subu $sp,12
  sw $fp,8($sp)
  sw $s0,4($sp)
  addiu $fp,$sp,8
    # $sp:(n)($s0 v)$fp:($fp v)
  li $v0,5       # read int
  syscall
  sw $v0,-8($fp) # scanf("%d",&n)
  li $s0,0       # i=0
  main_ciclu1_inceput:
  lw $t0,-8($fp) # $t0=n
  bge $s0,$t0,main_ciclu1_sfarsit
    jal getinthexa  # functie fara parametri, cu retur prin $v0
    la $t0,v        # $t0=v
    move $t1,$s0    # $t1=i
    sll $t1,$t1,2   # $t1=4*i
    add $t0,$t0,$t1 # $t0=(unsigned char *)v+4*i=v+i
    sw $v0,0($t0)   # v[i]=getinthexa();
  add $s0,1  # ++i
  b main_ciclu1_inceput
  main_ciclu1_sfarsit:
  li $s0,0       # i=0
  main_ciclu2_inceput:
  lw $t0,-8($fp) # $t0=n
  bge $s0,$t0,main_ciclu2_sfarsit
    li $v0,1        # print int
```

```
    move $t0,$s0    # $t0=i
    sll $t0,$t0,2   # $t0=4*i
    la $a0,v
    add $a0,$a0,$t0 # $a0=(unsigned char *)v+4*i=v+i
    lw $a0,0($a0)   # $a0=v[i]
    syscall
    li $v0,4        # print string
    la $a0,blank
    syscall
  add $s0,1  # ++i
  b main_ciclu2_inceput
  main_ciclu2_sfarsit:
  li $v0,4        # print string
  la $a0,nl
  syscall
    # iesirea din "main"
  li $v0,0       # "main" returneaza 0
  lw $s0,-4($fp) # restauram $s0
  lw $fp,0($fp)  # restauram $fp
  addu $sp,12    # eliminam cadrul de apel
jr $ra           # revenim la kernel (care a lansat programul cu "jal main")
getinthexa:
    # "getinthexa" e o functie ce necesita un cadru de apel
    #   in care alocam "buf" si salvam $fp, $s0 (va fi "i"), $s1 (va fi "n");
    # "buf" are dim. constanta (11 bytes pe care il vom aproxima prin 12
    #   bytes ca sa avem adrese aliniate), deci cadrul are dim. constanta
    #   (anume 4+4+4+12=24 bytes), nu ca in exemplul anterior
  subu $sp,24
  sw $fp,20($sp)
  sw $s0,16($sp)
  sw $s1,12($sp)
  addiu $fp,$sp,20
    # $sp:(buf[0]...buf[3])(buf[4]...buf[7])(buf[8]...buf[10],.)
    #        ($s1 v)($s0 v)$fp:($fp v)
  li $v0,8        # read string
  subu $a0,$fp,20 # $a0=buf
  li $a1,11       # lungimea maxima a sirului citit
  syscall         # gets(buf)
  li $s1,0        # n=0;
  li $s0,2        # i=2
  getinthexa_ciclu_inceput:
  li $t0,10
  bge $s0,$t0,getinthexa_ciclu_sfarsit
    subu $t0,$fp,20 # $t0=buf
    add $t0,$t0,$s0 # $t0=buf+i ("buf" are componente de 1 byte, nu 1 word)
    lb $t0,0($t0)   # $t0=buf[i] (byte)
    sub $t1,$t0,'0' # $t1=buf[i]-'0'
    li $t2,'a'
    blt $t0,$t2,getinthexa_et
      sub $t1,39    # if(buf[i]>='a') $t1=(buf[i]-'0')-39
    getinthexa_et:
    sll $s1,$s1,4   # n=n*16
    add $s1,$s1,$t1 # n=n*16+buf[i]-'0' sau n=n*16+(buf[i]-'0')-39
  add $s0,1       # ++i
  b getinthexa_ciclu_inceput
  getinthexa_ciclu_sfarsit:
    # iesirea din "getinthexa"
  move $v0,$s1    # valoarea returnata "n"
  lw $s0,-4($fp)
  lw $s1,-8($fp)
  lw $fp,0($fp)
  addu $sp,24
jr $ra
##########
```

Comentarii:
- programul MIPS de mai sus nu ruleaza corect pe PCSpim din cauza unor
 probleme legate de citirea de la consola; cel mai bine se poate urmari
 ruland pas cu pas (cu F10);
- intrucat vectorul local automatic "buf" are dimensiune fixa (anume 10
 bytes) alocarea lui se face obisnuit iar cadrul de apel are dimensiune fixa
 24 bytes (cunoscuta in faza de elaborare a programului), nu ca in exemplul
 anterior; pentru asemenea vectori, calculul adresei unei componente se poate
 face mai usor; nu s-a putut vedea asta in programul nostru deoarece "buf"
 era vector de byte, dar daca era vector de word operatia "$t0=buf[i]" se
 putea transcrie:

```
   move $t0,$s0    # $t0=i
   sll $t0,$t0,2   # $t0=4*i
   add $t0,$t0,$fp
   sub $t0,$t0,20  # $t0=4*i+$fp-20=(unsigned char *)buf+4*i=buf+i
   lw $t0,0($t0)   # $t0=buf[i]
```

 spre deosebire de operatia "$a0=v[i]" (unde se acceseaza un vector global)
 din "main" care necestia doi registri (nu putem folosi doar $a0):

```
   move $t0,$s0      # $t0=i
   sll $t0,$t0,2   # $t0=4*i
   la $a0,v
   add $a0,$a0,$t0 # $a0=(unsigned char *)v+4*i=v+i
   lw $a0,0($a0)   # $a0=v[i]
```

1k) Alte exemple: - TODO

 In exemplele care urmeaza vom respecta in general conventiile MIPS date in sectiunea 1j, cu exceptiile mentionate tot acolo. In particular, nu vom implementa "main" ca o procedura iar variabilele declarate local automatic in "main" vor fi alocate global (deci static). De asemenea, translatarea programelor C in MIPS nu va fi facuta intotdeauna fidel ci cu anumite optimizari.

Exemplu: Backtracking recursiv folosind un vector local static;
~~~~~~~~ programul genereaza permutarile de ordin n.

Translatam urmatorul program C:

```
#include<stdio.h>

void gen(int n){
 static int v[10],k=0;
 register int i,j;
 if(k==n){
   for(i=0;i<n;++i)printf("%d ",v[i]);
   printf("\n");
 }
 for(i=1;i<=n;++i){
   for(j=0;j<k;++j)if(v[j]==i)break;
   if(j==k){v[k]=i; ++k; gen(n); --k;}
 }
}

void main(){
  int n;
  scanf("%d",&n);
  gen(n);
}
```
~~~~~~~~

```
/* pe consola vedem (primul 3 e tastat de noi):
   3
   1 2 3
   1 3 2
   2 1 3
   2 3 1
   3 1 2
   3 2 1
*/
```

Translatarea este:

```
.data
  n: .space 4
.text
main:
    # citire "n"
  li $v0,5
  syscall
  sw $v0,n
    # apel "gen(n)"
  subu $sp,4
  lw $t0,n
  sw $t0,0($sp)
  jal gen
  addu $sp,4
li $v0,10
syscall
.data
  v: .space 40 # "v" desi local, se aloca static
  k: .word 0   # "k" e local static si se init. o data, inaintea apelurilor
  blank: .asciiz " "
  nl: .asciiz "\n"
.text
gen:
  subu $sp,16
  sw $ra,12($sp)
  sw $fp,8($sp)
  sw $s0,4($sp)
  sw $s1,0($sp)
  addiu $fp,$sp,12
    # $sp:($s1 v)($s0 v)($fp v)$fp:($ra v)(n)
    # convin ca $s0=i, $s1=j
  lw $t0,k
  lw $t1,4($fp)       # $t1=n
  bne $t0,$t1,gen_et1 # test k!=n
    li $s0,0            # i=0
    gen_et2:
    lw $t1,4($fp)       # $t1=n
    bge $s0,$t1,gen_et3 # test i>=n
      li $v0,1          # print int
      la $a0,v
      move $t0,$s0
      sll $t0,$t0,2
      add $a0,$a0,$t0
      lw $a0,0($a0)     # $a0=v[i]
      syscall
      li $v0,4          # print string
      la $a0,blank
      syscall
    add $s0,1           # ++i
    b gen_et2
    gen_et3:
    li $v0,4  # print string
```

```
    la $a0,nl
    syscall
  gen_et1:
  li $s0,1             # i=1
  gen_et4:
  lw $t1,4($fp)        # $t1=n
  bgt $s0,$t1,gen_et5  # test i>n
    li $s1,0             # j=0
    lw $t2,k             # $t2=k
    la $t3,v             # $t3=adr.lui v
    gen_et6:
    bge $s1,$t2,gen_et7  # test j>=k
      move $t0,$s1
      sll $t0,$t0,2
      add $t0,$t0,$t3
      lw $t0,0($t0)        # $t0=v[j]
      beq $t0,$s0,gen_et7  # if(v[j]==i)break;
    add $s1,1            # ++j
    b gen_et6
    gen_et7:
    bne $s1,$t2,gen_et8  # test j!=k ($t2 inca contine k)
      move $t0,$t2    # $t0=k
      sll $t0,$t0,2   # $t0=4*k
      add $t0,$t0,$t3 # $t0=(unsigned char*)v+4*k=v+k ($t3 inca contine v)
      sw $s0,0($t0)   # v[k]=i
      add $t2,1
      sw $t2,k        # ++k
      subu $sp,4
      lw $t0,4($fp)
      sw $t0,0($sp)   # push n
      jal gen         # gen(n) (apelul imbricat)
      addu $sp,4
      lw $t2,k
      sub $t2,1
      sw $t2,k        # --k
      lw $t3,v        # acum $t2, $t3 iar contin k, resp. v
    gen_et8:
  add $s0,1            # ++i
  b gen_et4
  gen_et5:
  lw $s1,-12($fp)
  lw $s0,-8($fp)
  lw $ra,0($fp)
  lw $fp,-4($fp)
  addu $sp,16
jr $ra
##########
```

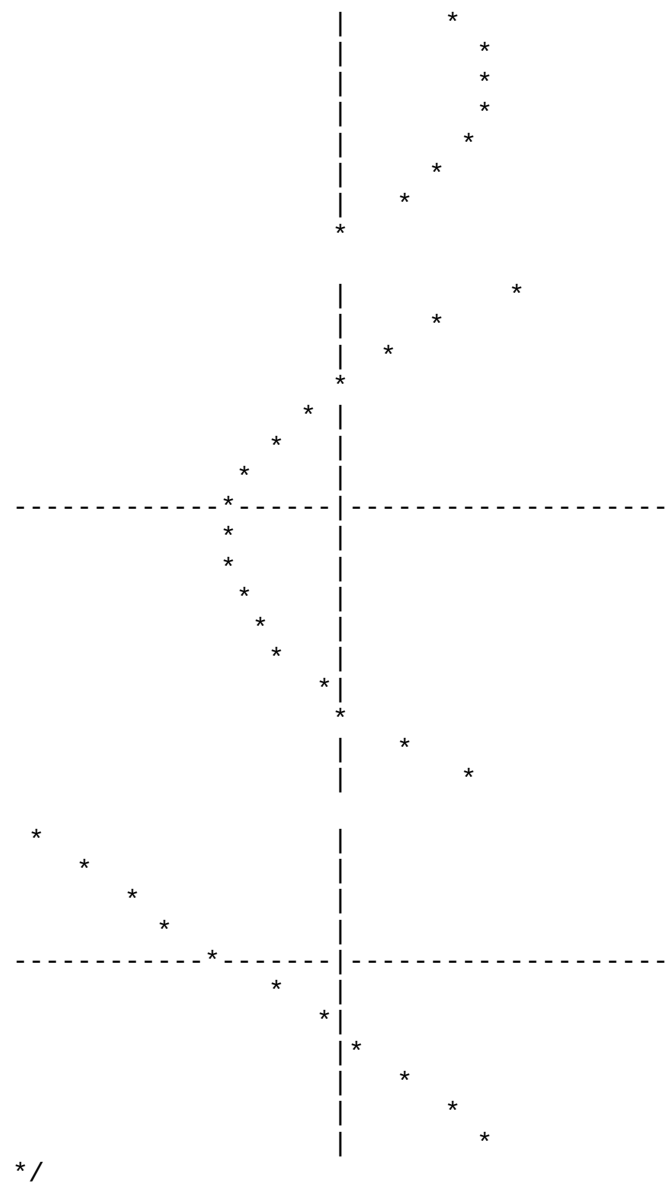
Comentarii:
- "v" si "k" sunt locale lui "gen" (pot fi accesate doar din "gen") dar sunt statice (deci se aloca in zona de date statice si isi pastreaza locatia pe toata perioada executarii programului); astfel o valoare lasata de un apel al lui "gen" in "v" sau "k" va fi regasita de apelul urmator (deci variabilele locale statice sunt un mijloc de comunicare intre apelurile succesive ale procedurii); in plus initializarea "k=0" are loc o singura data, inaintea primului apel al lui "gen" (nu la fiecare nou apel);
- desi la prima intalnire a lui "gen_et2:" $t1 contine deja "n", l-am reincarcat cu "lw $t1,4($fp)" deoarece aici ajungem si dupa ce s-au facut iteratii, iar in iteratii facem "syscall" care ar putea altera $t1 (nu e callee-saved); la fel am procedat si la "gen_et4:"; de asemenea, $t2 si $t3 incarcate inainte de "gen_et6:" cu "k", respectiv adresa lui "v" isi pastreaza valoare (deoarece nu facem apeluri imbricate sau "syscall") pana la apelul imbricat "gen(n)", deci nu mai trebuie reincarcati la fiecare calcul "v[i]" sau "v[k]"; cand facem "++k" sau "--k" in preajma apelului

imbricat "gen(n)" ei trebuie reincarcati, inclusiv "k" trebuie salvat in memorie, deoarece $t2, $t3 nu sunt callee-saved.

Exemplu: Parametri (pointeri la) functii;
~~~~~~~~ programul reprezinta grafic (in mod text) niste functii reale de variabila reala, folosind o procedura care primeste adresele lor ca parametri.

Translatam urmatorul program C:

```
#include<stdio.h>

void reprezinta(double (*f)(double),double a,double b,double dx,double dy){
  char linie[42];
  register int i,k; register char c; register double x,y;
  for(x=a;x<=b;x+=dx){
    y=(*f)(x); k=(int)(y/dy)+20;
    if(0<=x && x<dx)c='-'; else c=' ';
    for(i=0;i<=40;++i)linie[i]=c;
    linie[20]='|';
    if(0<=k && k<=40)linie[k]='*';
    linie[41]='\0';
    printf("%s\n",linie);
  }
}

double sin(double x) /* aproximeaza sin(x) */
 {return x-x*x*x/6+x*x*x*x*x/120-x*x*x*x*x*x*x/5040;}

double parabola(double x) /* o parabola */
 {return 0.3*x*x-0.3*x-0.7;}

double dreapta(double x) /* o dreapta */
 {return x-1;}

double (*vect_f[])(double)= {sin, parabola, dreapta};
double vect_a[]= {  -3,  -2,  -1};
double vect_b[]= {   3,   3,   2};
double vect_dx[]={ 0.3, 0.3, 0.3};
double vect_dy[]={ 0.1, 0.1, 0.1};
int n=3;

void main(){
  register int i;
  for(i=0;i<n;++i){
    reprezinta(vect_f[i],vect_a[i],vect_b[i],vect_dx[i],vect_dy[i]);
    printf("\n");
  }
}

/* afisaza:
                     *
                  *  |
                *    |
              *      |
            *        |
           *         |
           *         |
           *         |
             *       |
               *     |
                  *  |
                     *
--------------------|-*------------------
                    |    *
```
~~~~~~~~




```
*/

Translatarea este:

.data
  vect_f: .word sin, parabola, dreapta
  vect_a: .double -3.0, -2.0, -1.0
  vect_b: .double  3.0,  3.0,  2.0
  vect_dx:.double  0.3,  0.3,  0.3
  vect_dy:.double  0.1,  0.1,  0.1
  n: .word 3
  nl: .asciiz "\n"
.text
##### programul principal
main:
  li $s0,0             # i=0
  main_et1:
  lw $t0,n
  bge $s0,$t0,main_et2 # test i>=n
    sll $t0,$s0,2  # $t0=4*i
    sll $t1,$s0,3  # $t1=8*i
    subu $sp,40
    la $t2,vect_dy
    add $t2,$t2,$t1
    l.d $f0,0($t2) # $f0=vect_dy[i]
    s.d $f0,32($sp)# push vect_dy[i]
```

```
    la $t2,vect_dx
    add $t2,$t2,$t1
    l.d $f0,0($t2) # $f0=vect_dx[i]
    s.d $f0,24($sp)# push vect_dx[i]
    la $t2,vect_b
    add $t2,$t2,$t1
    l.d $f0,0($t2) # $f0=vect_b[i]
    s.d $f0,16($sp)# push vect_b[i]
    la $t2,vect_a
    add $t2,$t2,$t1
    l.d $f0,0($t2) # $f0=vect_a[i]
    s.d $f0,8($sp) # push vect_a[i]
    la $t2,vect_f
    add $t2,$t2,$t0
    lw $t3,0($t2) # $f0=vect_f[i]
    sw $t3,0($sp) # push vect_f[i]
    jal reprezinta
    addu $sp,40
    li $v0,4  # print string
    la $a0,nl
    syscall   # printf("\n");
  add $s0,1            # ++i
  b main_et1
  main_et2:
li $v0,10
syscall
##### procedura "reprezinta"
reprezinta:
# primeste: $sp:(f (pointer))()[a][b][dx][dy]
# unde () inseamna un word iar [] inseamna un dublu word
  subu $sp,88
  sw $ra,84($sp)
  sw $fp,80($sp)
  s.d $f0,72($sp) # $f0 va fi x (adresa este multiplu de 8)
  s.d $f2,64($sp) # $f2 va fi un reg. auxiliar (adresa este multiplu de 8)
  s.d $f4,56($sp) # $f4 va fi un reg. auxiliar (adresa este multiplu de 8)
  sw $s2,52($sp)  # $s2 va fi c
  sw $s1,48($sp)  # $s1 va fi k
  sw $s0,44($sp)  # $s0 va fi i
  addiu $fp,$sp,84
    # $sp:(linie[0]...linie[3])...(linie[40],linie[41],.,.)
    #        ($s0 v)($s1 v)($s2 v)[$f4 v][$f2 v][$f0 v]($fp v)$fp:($ra v)
    #        (f)()[a][b][dx][dy]
  l.d $f0,12($fp)     # x=a
  reprezinta_et1:
  l.d $f2,20($fp)     # $f2=b
  c.lt.d $f2,$f0      # daca x>b, flag-ul 0 al coprocesorului 1 devine 1
  bc1t reprezinta_et2 # daca flag-ul 0 al coprocesorului 1 este 1, salt
    subu $sp,8
    s.d $f0,0($sp)     # push x
    lw $t0,4($fp)      # $t0=f
    jalr $ra,$t0       # apel (*f)(x) (returneaza un double prin stiva)
    l.d $f2,0($sp)     # $f2=(*f)(x)=y
    addu $sp,8
    l.d $f4,36($fp)    # $f4=dy
    div.d $f2,$f2,$f4  # $f2=y/dy
    cvt.w.d $f4,$f2    # $f4=(int)(y/dy), codificat ca int (!)
    mfc1.d $t0,$f4     # copiaza config. din ($f4,$f5) in ($t0,$t1)
    move $s1,$t0       # k=(int)(y/dy) (eventual trunchiat)
    li $t0,20
    add $s1,$s1,20     # k=(int)(y/dy)+20
    li $t0,0           # $t0=false
    li $t2,1           # $t2=true
    li.d $f2,0.0       # $f2=0
```

```
    c.le.d $f2,$f0     # flag-ul 0 al cop. 1 devine 1 daca 0<=x, si 0 altfel
    mtc1.d $t0,$f2     # $f2=false
    mtc1.d $t2,$f4     # $f4=true
    movt.d $f2,$f4,0   # daca flag-ul 0 e 1, $f2 ia val. $f4, adica true
    mfc1.d $t4,$f2     # $t4=val. de adevar a lui "0<=x"
    l.d $f2,28($fp)    # $f2=dx
    c.le.d $f0,$f2     # flag-ul 0 al cop. 1 devine 1 daca x<=dx,si 0 altfel
    mtc1.d $t0,$f2     # $f2=false
    mtc1.d $t2,$f4     # $f4=true
    movt.d $f2,$f4,0   # daca flag-ul 0 e 1, $f2 ia val. $f4, adica true
    mfc1.d $t6,$f2     # $t6=val. de adevar a lui "x<=dx"
    and $t4,$t4,$t6    # $t4=val. de adevar a lui "0<=x && x<=dx"
    li $s2,'-'         # c='-'
    bnez $t4,reprezinta_et3 # daca "0<=x && x<=dx" e adev. "c" ramane '-'
    li $s2,' '
    reprezinta_et3:
    li $s0,0                    # i=0
    reprezinta_et4:
    li $t0,40
    bgt $s0,$t0,reprezinta_et5  # test i>40
      addiu $t0,$fp,-84  # $t0=adr.lui "linie"
      add $t0,$t0,$s0    # $t0=adr.lui "linie[i]" (e byte, nu trebuie *4)
      sb $s2,0($t0)      # linie[i]=c
    add $s0,1
    b reprezinta_et4
    reprezinta_et5:
    addiu $t0,$fp,-84  # $t0=adr.lui "linie"
    addiu $t0,$t0,20   # $t0=adr.lui "linie[20]"
    li $t1,'|'
    sb $t1,0($t0)      # linie[20]='|'
    sle $t0,$zero,$s1  # $t0 = (0<=k)
    li $t1,40
    sle $t1,$s1,$t1    # $t1 = (k<=40)
    and $t0,$t0,$t1    # $t0 = (0<=k && k<=40)
    beqz $t0,reprezinta_et6 # daca 0<=k && k<=40 e fals, nu pun "*"
      li $t1,'*'
      addiu $t0,$fp,-84  # $t0=adr.lui "linie"
      add $t0,$t0,$s1    # $t0=adr.lui "linie[k]"
      sb $t1,0($t0)      # linie[k]='*'
    reprezinta_et6:
    addiu $t0,$fp,-84  # $t0=adr.lui "linie"
    addiu $t0,$t0,41   # $t0=adr.lui "linie[41]"
    sb $zero,0($t0)    # linie[41]='\0'
    li $v0,4           # print string
    addiu $a0,$fp,-84  # $a0=adr.lui "linie"
    syscall
    li $v0,4
    la $a0,nl
    syscall
  l.d $f2,28($fp)     # $f2=dx
  add.d $f0,$f0,$f2   # x+=dx
  b reprezinta_et1
  reprezinta_et2:
  l.d $f0,-12($fp)
  l.d $f2,-20($fp)
  l.d $f4,-28($fp)
  lw $s2,-32($fp)
  lw $s1,-36($fp)
  lw $s0,-40($fp)
  lw $ra,0($fp)
  lw $fp,-4($fp)
  addu $sp,88
jr $ra
##### functia "sin"
```

```
sin: # primeste $sp:[x] si returneaza prin stiva
  subu $sp,40
  sw $fp,32($sp)
  s.d $f6,24($sp)
  s.d $f4,16($sp)
  s.d $f2,8($sp)
  s.d $f0,0($sp)
  addiu $fp,$sp,36 # $sp:[$f0 v][$f2 v][$f4 v][$f6 v]($fp v)$fp:()[x]
  l.d $f0,4($fp)     # $f0=x
  mov.d $f6,$f0      # $f6=x
  mul.d $f2,$f0,$f0  # $f2=x*x
  mul.d $f2,$f2,$f0  # $f2=x*x*x
  li.d $f4,6.0
  div.d $f4,$f2,$f4  # $f4=x*x*x/6
  sub.d $f6,$f6,$f4  # $f6=x-x*x*x/6
  mul.d $f2,$f2,$f0  # $f2=x*x*x*x
  mul.d $f2,$f2,$f0  # $f2=x*x*x*x*x
  li.d $f4,120.0
  div.d $f4,$f2,$f4  # $f4=x*x*x*x*x/120
  add.d $f6,$f6,$f4  # $f6=x-x*x*x/6+x*x*x*x*x/120
  mul.d $f2,$f2,$f0  # $f2=x*x*x*x*x*x
  mul.d $f2,$f2,$f0  # $f2=x*x*x*x*x*x*x
  li.d $f4,5040.0
  div.d $f4,$f2,$f4  # $f4=x*x*x*x*x*x*x/5040
  sub.d $f6,$f6,$f4  # $f6=x-x*x*x/6+x*x*x*x*x/120+x*x*x*x*x*x*x/5040
  s.d $f6,4($fp)   # $sp:[$f0 v][$f2 v][$f4 v][$f6 v]($fp v)$fp:()[rezultat]
  l.d $f6,-12($fp)
  l.d $f4,-20($fp)
  l.d $f2,-28($fp)
  l.d $f0,-36($fp)
  lw $fp,-4($fp)
  addu $sp,40
jr $ra
##### functia "parabola"
parabola: # primeste $sp:[x] si returneaza prin stiva
  subu $sp,40
  sw $fp,32($sp)
  s.d $f6,24($sp)
  s.d $f4,16($sp)
  s.d $f2,8($sp)
  s.d $f0,0($sp)
  addiu $fp,$sp,36 # $sp:[$f0 v][$f2 v][$f4 v][$f6 v]($fp v)$fp:()[x]
  l.d $f0,4($fp)     # $f0=x
  mul.d $f2,$f0,$f0  # $f2=x*x
  li.d $f4,0.3       # $f4=0.3
  mul.d $f6,$f4,$f2  # $f6=0.3*x*x
  mul.d $f2,$f4,$f0  # $f2=0.3*x
  sub.d $f6,$f6,$f2  # $f6=0.3*x*x-0.3*x
  li.d $f2,0.7
  sub.d $f6,$f6,$f2  # $f6=0.3*x*x-0.3*x-0.7
  s.d $f6,4($fp)   # $sp:[$f0 v][$f2 v][$f4 v][$f6 v]($fp v)$fp:()[rezultat]
  l.d $f6,-12($fp)
  l.d $f4,-20($fp)
  l.d $f2,-28($fp)
  l.d $f0,-36($fp)
  lw $fp,-4($fp)
  addu $sp,40
jr $ra
##### functia "dreapta"
dreapta: # primeste $sp:[x] si returneaza prin stiva
  subu $sp,24
  sw $fp,16($sp)
  s.d $f2,8($sp)
  s.d $f0,0($sp)
```

```
  addiu $fp,$sp,20 # $sp:[$f0 v][$f2 v]($fp v)$fp:()[x]
  l.d $f0,4($fp)     # $f0=x
  li.d $f2,1.0
  sub.d $f2,$f0,$f2
  s.d $f2,4($fp)   # $sp:[$f0 v][$f2 v]($fp v)$fp:()[rezultat]
  l.d $f2,-12($fp)
  l.d $f0,-20($fp)
  lw $fp,-4($fp)
  addu $sp,24
jr $ra
##########

Comentarii:
- procedura "reprezinta" reprezinta grafic in mod text, sub forma unui sir
 de "*" pe verticala, desenand si axele de coordonate (Ox pe verticala, Oy
 pe orizontala), o functie reala de variabila reala a carei adresa este
 primita ca parametru; semnificatia parametrilor lui "reprezinta" este:
 "f": adresa functiei reale de variabila reala ce trebuie reprezentata;
 "a", "b": capetele intervalului de definitie a functiei;
 "dx", "dy": scara de reprezentare; practic "dx", "dy" inseamna lungimea
   segmentului pe Ox, resp. Oy, care pe ecran se reprezinta printr-un
   singur caracter;
 procedura construieste graficul linie cu linie in stringul "linie" si
 fiecare linie construita este imediat afisata; fiecare asemenea linie este
 o linie verticala corespunzatoare cate unei valori a lui "x" in intervalul
 ["a","b"]; acest interval este parcurs in pasi de lungime "dx" (ei
 corespund unui salt de un caracter pe ecran); intrucat graficul este
 desenat pe verticala, liniile verticale construite succesiv sunt afisate
 orizontal (cu "printf("%s\n",linie)");
  "y" este imaginea lui "x" in multimea numerelor reale iar "k"
 corespondentul lui "y" pe ecran, in numar de caractere fata de caracterul
 origine;
  cel mult una din liniile afisate poate contine axa Oy; acest lucru este
 detectat cu conditia "0<=x && x<dx"; la jumatatea fiecarei linii trebuie
 desenata intersectia cu Ox, sub forma unui '|';
- in programul C, pe langa procedura "reprezinta", am definit un vector de
 pointeri la functii (ce primesc un parametru double si returneaza double)
 "vect_f" si niste vectori de double "vect_a","vect_b","vect_dx","vect_dy",
 pe care i-am initializat cu adresele unor functii "sin", "parabola",
 "dreapta" si niste valori a, b, dx, dy potrivite pentru fiecare; in "main"
 am aplicat "reprezinta" pentru fiecare asemenea sistem;
- in programul MIPS am tinut cont ca pointerii sunt pe 4 octeti iar valorile
 double pe 8 octeti - asta a influentat de ex. regula de calculare a
 offset-urilor componentelor vectorilor de pointeri si double si spatiul
 alocat in stiva pentru ele;
  de asemenea, intrucat in stiva se salveaza si valori double (parametri,
 registrii $f0, $f2 salvati/restaurati) iar acestea nu se pot scrie/citi
 decat la adrese multiplu de 8 (dimensiunea tipului double), am avut grija
 ca $sp sa ia doar valori multiplu de 8; de aceea in "main" am push-at
 parametrul "vect_f[i]" tot pe 8 octeti, desi valoarea lui are doar 4
 octeti; puteam sa nu constrang $sp sa varieze din 8 in 8, dar atunci
 trebuia sa calculez si sa aloc spatii suplimentare la scrierea in stiva
 a fiecarui double;
- functiile "sin", "parabola", "dreapta" au fost implementate sa returneze
 prin stiva;
- pentru instructiunile in virgula mobila a se (re)vedea lectia 2;
 notam ca flag-ul 0 al coprocesorului 1 setat cu "c.lt.d", "c.le.d",etc. isi
 pastreaza valoarea pana se executa o alta instructiune de acest tip (adica
 despre care am specificat in lectia 2 ca modifica flag-ul);
- pentru evaluarea expresiei "0<=x && x<=dx" am fi avut nevoie, pentru a
 evita multiplele ramificari, de instructiuni de tip "sle" care sa accepte
 operanzi in virgula mobila; intrucat asemenea instructiuni nu au fost
 gasite in cartea [1], am simulat efectul lor cu niste artificii; de ex.
 pentru a evalua "$t4 = 0<=x", adica "$t4 = 0<=$f0" am efectuat:
```

```
    li $t0,0           # $t0=false
    li $t2,1           # $t2=true
    li.d $f2,0.0       # $f2=0
    c.le.d $f2,$f0     # flag-ul 0 al cop. 1 devine 1 daca 0<=x, si 0 altfel
    mtc1.d $t0,$f2     # $f2=false
    mtc1.d $t2,$f4     # $f4=true
    movt.d $f2,$f4,0   # daca flag-ul 0 e 1, $f2 ia val. $f4, adica true
    mfc1.d $t4,$f2     # $t4=val. de adevar a lui "0<=x"
```

echivalent, in locul ultimelor doua instructiuni am fi putut efectua:

```
    movf.d $f4,$f2,0   # daca flag-ul 0 e 0, $f4 ia val. $f2, adica false
    mfc1.d $t4,$f4     # $t4=val. de adevar a lui "0<=x"
```

ne-am bazat pe faptul ca "mtc1.d" si "mfc1.d" copiaza configuratiile, nu schimba modul de reprezentare a valorilor (deci din $t-uri se copiaza in $f-uri 1 si 0 ca intregi, apoi inapoi in $t4 tot ca intregi);
 mentionam ca "mtc1.d" si "mfc1.d" afecteaza de fapt perechile ($t0,$t1), ($t2,$t3), resp. ($t4,$t5) pe de-o parte si ($f2,$f3), resp. ($f4,$f5) pe de alta parte; dar, indiferent ce contin sau vor contine $t1, $t3, $t5, $f3, $f4 tot ce ne intereseaza se transfera doar intre $t0, $t2, $t4, $f2, $f4; de aceea de ex. nu a trebuit sa initializam si $t1, $t3;
- in cartea [1] nu am gasit conventii MIPS referitoare la salvarea/restaurarea registrilor coprocesorului 1 cu ocazia apelurilor de proceduri; in absenta unor asemenea conventii i-am considerat callee-saved;
- in procedura "reprezinta" trebuie salvat/restaurat $ra, deoarece face apeluri imbricate (prin acel "(*f)(x)"); in "sin", "parabola" si "dreapta" nu e necesar, dar am rezervat un word in plus in cadru pentru a pastra alinierea adreselor la multipli de 8.

2. Macro-uri: - TODO

3. Exercitii: - TODO

In toate cazurile, in afara celor mentionate explicit, parametrii procedurilor se vor pasa prin stiva, valoarea returnata (in cazul functiilor) se va intoarce tot prin stiva,iar macro-urile nu se vor rescrie in proceduri.
Programele marcate cu (*) se vor realiza in clasa.

III.1) (puncte - vezi in text) (*)
a) (1 punct)
  Functie "sign" ce primeste un parametru word x si returneaza (prin $v0) valoarea 1 daca x>0, 0 daca x=0, -1 daca x<0. Program ilustrativ.
b) (1 punct)
  Acelasi lucru ca la (a) dar cu retur prin stiva.
c) (1 punct)
  Acelasi lucru ca la (a) dar cu un macro.

III.2) (puncte - vezi in text) (*)
a) (1.5 puncte)
  Functie MIPS ce implementeaza functia C "memcpy":

```
     void *memcpy(void *dest, const void *src, size_t n);
```

care copiaza a "n" octeti de la adresa "src" la adresa "dest" si returneaza "dest" (putem considera "src" si "dest" adrese obisnuite si "n" word).
Program ilustrativ care copiaza continutul unui vect. de word-uri in altul.
b) (1 punct)
  Acelasi lucru cu un macro.

III.3) (2 puncte) (*)

Translatati in MIPS urmatorul program C (el contine o procedura ce afisaza frecventa valorilor dintr-un vector de numere de o cifra):

```
#include<stdio.h>

void frecv(int *v, int n){
  int f[10], k;     /* var. k si vect. f se vor aloca automatic pe stiva */
  register int i,j; /* i si j se vor aloca in registri */
  for(i=0;i<10;++i) f[i]=0;
  for(i=0;i<n;++i) if(0<=v[i] && v[i]<10) ++f[v[i]];
  k=0; for(i=0;i<10;++i) if(f[i]!=0)++k;
  for(i=0;i<10;++i){
    printf("%d",f[i]);    /* se va folosi syscall, functia print int */
    printf(" ");          /* se va folosi syscall, functia print string */
  }
  printf("%d",k);         /* se va folosi syscall, functia print int */
  printf("\n");           /* se va folosi syscall, functia print string */
}

int x[7]={1,2,3,3,3,1,5};

void main(){
  frecv(x,7);
}

/* se va afisa: 0, 2, 1, 3, 0, 1, 0, 0, 0, 0, 4
   deoarece 0 apare de 0 ori, 1 de 2 ori, 2 o data, 3 de 3 ori, ...,
   9 de 0 ori si sunt 4 frecvente nenule */
```

III.4) (puncte - vezi in text) (*)

a) (3.5 puncte)
Functie C(n,k) recursiva ce calculeaza combinari de n luate cate k dupa formula: C(0,0) = C(1,0) = 0, C(n,k) = C(n,k-1) + C(n-1,k-1). Program ilustrativ.

b) (3.5 puncte)
Acelasi lucru cu un macro recursiv (si retur prin $v0).

III.5) (3.5 puncte)
Functie ce primeste ca parametru un numar natural n si un caracter c si returneaza un string (construit in stiva) ce contine de n ori caracterul c si un caracter nul la sfarsit.
Program ilustrativ in care n si c sunt date prin variabile initializate, iar in programul principal se apeleaza functia, apoi stringul returnat este extras din stiva si copiat intr-o variabila statica de tip .space (inclusiv caracterul nul de la sfarsit), apoi este afisat la consola.

III.6) (1.5 puncte)
Translatati in MIPS urmatorul program C (respectand conventiile C):

```
#include<stdarg.h>
void aduna(int *a, int n, ...){
  register int i;
  va_list l;
  va_start(l,n);
  *a=0;
  for(i=0;i<n;++i) *a+=va_arg(l, int);
  va_end(l);
}
int s, s1;
void main(){
  aduna(&s,3,1,2,3);    /* obtinem s=6 */
  aduna(&s1,2,10,20);   /* obtinem s1=30 */
}
```

(i nu va fi alocat automatic ci pentru el se va folosi un registru).

III.7) (1.5 puncte)
Scrieti o functie pentru adunarea unui sir de numere, cu numar variabil de parametri (parametrii sunt numarul numerelor si apoi numerele) si retur prin stiva. Program ilustrativ. Indicatie: adaptam programul de la III.6.

III.8) (3 puncte) (*)
Translatati in MIPS urmatorul program C:

```
int aplica(int (*f)(int), int x){
  return (*f)(x);
}

int f1(int y){return y+y;}
int f2(int y){return y*y;}
int f3(int y){return -y;}

int (*vf[])={f1, f2, f3}, v[3];

void main(){
  register int i;
  for(i=0;i<3;++i) v[i]=aplica(vf[i],1+i);
}

/* in final v[0]=2, v[1]=4, v[2]=-3 */
```

III.9) (1.5 puncte) (*)
Translatati in MIPS urmatorul program C (ce contine o procedura care pune bitii dintr-un intreg intr-un string sub forma de caractere '1' sau '0'):

```
#include<stdio.h>

void bts(int n, char *s){
  register int i;
  for(i=0;i<8*sizeof(int);++i) if(n & (1<<i)) s[i]='1'; else s[i]='0';
  s[8*sizeof(int)]='\0';
   /* in MIPS in loc de 8*sizeof(int) vom pune 32 */
}

void main(){
  int x=259;
  char y[100];      /* in MIPS vom pune .space 100 */
  bts(x,y);
  printf("%s\n",y); /* se va folosi syscall, functia print string */
}

/* in MIPS se va afisa: 11000000100000000000000000000000 */
```

III.10) (puncte - vezi in text) (*)
a) (3.5 puncte)
Functie MIPS ce implementeaza functia C "atoi" (cu retur prin $v0):

```
  int atoi(const char *s);
```

care returneaza intregul a carui reprezentare externa zecimala este continuta in stringul "s" (putem considera "s" o adresa obisnuita); presupunem stringul "s" corect construit (si terminat prin caracterul nul); de exemplu atoi("12") returneaza int-ul 12. Program ilustrativ.
b) (1 punct)
Acelasi lucru cu un macro.

III.11) (puncte - vezi in text) (*)

a) (3 puncte)
  Procedura pentru sortarea unui vector de word prin metoda Bubble sort;
   procedura primeste ca parametri adresa de inceput a vectorului si numarul
   elementelor sale. Program ilustrativ.
b) (1 punct)
  Acelasi lucru cu un macro.

III.12) (1.5 puncte) (*)
 Functie recursiva ce returneaza (in $v0) suma cifrelor in baza 10 ale unui
word dat ca parametru.
 Ideea recursiei: s(0) = 0, altfel s(n) = n mod 10 + s(n div 10).
 Program ilustrativ.

III.13) (puncte - vezi in text) (*)
a) (1 punct)
  Functie MIPS ce implementeaza functia C "strlen" (cu retur prin $v0):

    size_t strlen(const char *s);

 care returneaza lungimea stringului pointat de "s", fara a numara
 terminatorul nul (putem considera "s" adresa obisnuita iar valoarea
 returnata int).
  Program ilustrativ.
b) (1 punct)
  Acelasi lucru cu un macro.

III.14) (puncte - vezi in text) (*)
a) (2.5 puncte)
  Functie MIPS ce implementeaza functia C "strcmp" (cu retur prin $v0):

    int strcmp(const char *s1, const char *s2);

 care compara lexicografic doua stringuri (pointate de "s1" si "s2") si
 returneaza o valoare <0 daca primul string este < al doilea, ==0 daca
 primul string este == al doilea si >0 daca primul string este > al doilea
 (putem considera "s1" si "s2" adrese obisnuite).
  Program ilustrativ.
b) (1 punct)
  Acelasi lucru cu un macro.

III.15) (puncte - vezi in text) (*)
a) (2 puncte)
  Functie MIPS ce implementeaza functia C "strrev" (fara retur):

    char *strrev(char *s);

 care inverseaza ordinea caracterelor din stringul pointat de "s", in afara
 de terminatorul nul (putem considera "s" adresa obisnuita iar valoarea
 returnata void).
   Program ilustrativ.
b) (1 punct)
  Acelasi lucru cu un macro.

III.16) (puncte - vezi in text) (*)
a) (1.5 puncte)
  Procedura ce primeste ca parametri adresele de inceput a doua stringuri
   si il copiaza pe primul in al doilea inlocuind toate literele mici cu mari
   (restul caracterelor se copiaza neschimbate); stringul sursa se considera
   terminat cu 0. Program ilustrativ.
b) (1 punct)
  Acelasi lucru ca la (a) dar cu un macro.
c) (3 puncte)
  Acelasi lucru ca la (a) dar procedura are doar un parametru, pt. stringul
   sursa, stringul destinatie este returnat prin stiva.

III.17) (puncte - vezi in text) (*)
a) (1.5 puncte)
  Functie (nerecursiva) ce primeste ca parametri adresa de inceput si nr. elementelor unui vector de word si returneaza max. elementelor sale. Program ilustrativ.
b) (3 puncte)
  Acelasi lucru ca la (a) dar cu o functie recursiva; ideea recursiei:
   * daca vectorul are un element, el este max.;
   * daca vectorul are mai multe elemente, se imparte in doua, se afla max. din fiecare jumatate (apeland recursiv functia) si se ret. max. celor doua max.;
  functia primeste ca par. adresa de inceput a bucatii analizate si lungimea ei si ret. max. elementelor din bucata respectiva.
c) (1 punct)
  Acelasi lucru ca la (a) dar cu un macro.

III.18) (puncte - vezi in text) (*)
a) (2 puncte)
  Procedura recursiva ce transforma un vector de word a.i. la inceput sa apara elem. sale impare in aceeasi ordine ca in vecorul initial, apoi elem. sale pare in ordinea inversa fata de vectorul initial. Parametri: adr. de inceput a vect. sursa, adr. vect. destinatie, nr. de elem. ale vect. sursa. Ideea recursiei: se foloseste o var. locala automatica; la fiecare apel se ia cate un word din vectorul sursa si:
   - daca e impar se scrie in vectorul destinatie, apoi se apeleaza recursiv pt. restul vectorului sursa;
   - daca e par se retine in var. locala apelului, se apeleaza recursiv pt. restul vectorului sursa, apoi (la revenirea din recursie) se scrie in vect. destinatie valoarea salvata.
  Ex: 1, 2, 7, 8, 2, 5, 6, 1, 5, 4, 3 --> 1, 7, 5, 1, 5, 3, 4, 6, 2, 8, 2
b) (2 puncte)
  Acelasi lucru cu un macro recursiv.

III.19) (8 puncte) (*)
  Procedura ce transforma o expresie din forma infixata in forma sufixata (forma poloneza inversa); expresia poate contine operanzi litera mica, operatori aditivi '+', '-', operatori multiplicativi '*', '/' si paranteze '(', ')'; toti operatorii sunt binari, cei aditivi avand prioritate mai mica decat cei multiplicativi; exemplu de transformare:

   sursa: '(a+b)*(a-c)+a' -> 'ab+ac-*a+'

  expresia sursa se considera data intr-un string declarat cu initializare, cea destinatie intr-un string declarat fara initializare (.space); expresia sursa se considera scrisa corect; procedura va opera asupra celor doua stringuri ca date globale si va primi ca parametru nivelul prioritatii (adica daca la apelul curent se trateaza o operatie aditiva, multiplicativa, etc.).
  Ideea recursiei urmareste definitia recursiva a unei expresii infixate:

  <expresie> ::= <termen> | <termen> <op_aditiva> <expresie>
  <termen>   ::= <factor> | <factor> <op_multiplicativa> <termen>
  <factor>   ::= <litera> | (<expresie>)

  in functie de parametrul - nivel procedura va urmari cate una din cele trei definitii.
  Indicatie: sirul sursa se citeste liniar, caracter cu caracter, folosind o variabila globala "c"; la fiecare apel se va procesa cate un car. "c" din sirul sursa; acest caracter este "citit" in codul apelant (deci primul caracter este citit in programul principal); la sfarsitul unui apel se va citi caracterul care se va procesa dupa revenire; de exemplu un apel de nivel 1 (expresie) va proceda astfel:
  - apel recursiv cu nivel 2 (caracterul curent "c", citit deja, se transm.

apelului imbricat - intr-adevar, primul caracter dintr-o expresie este primul caracter din primlul termen al expresiei); conform conventiei, la iesirea din apelul imbricat este deja citit noul "c";
- daca "c" este '+' sau '-', se salveaza intr-o var. locala automatica "k" (fiecare apel are deci propria instanta a lui "k"), se mai citeste un "c", se apeleaza recursiv cu acelasi nivel 1, la revenire se scrie "k" in sirul destinatie (apelul imbricat va furniza si un nou car. "c"), apoi se iese;
- daca nu, se iese (si "c" va fi prelucrat in alta parte).

In loc de o variabila globala "c" se poate parcurge sirul sursa cu un registru; sirul destinatie se poate parcurge de asemenea cu un registru (el este construit tot liniar).
Program ilustrativ.

III.20) (8 puncte)
Program pentru sortarea unui vector de word (declarat cu initializare in program) prin metoda Qsort, folosind o procedura recursiva; interactiunea dintre procedura si program se va face doar prin parametri/valoare returnata (nu prin variabile globale).

III.21) (1.5 puncte)
Procedura de interclasare a doi vectori sortati de word; parametri: adresele de inceput ale celur doi vectori sursa si a vectorului destinatie si nr. elementelor celor doi vectori sursa (deci 5 parametri); functia returneaza (prin $v0) nr. elementelor vectorului destinatie. Program ilustrativ.

III.22) (puncte - vezi in text) (*)
a) (1.5 puncte)
Functie MIPS ce implementeaza functia C "strset" (cu retur prin $v0):

```
char *strset(char *s, int ch);
```

care seteaza toate caracterele din stringul pointat de "s", in afara de terminatorul nul, la valoarea "ch", apoi returneaza "s" (putem considera "s" si returul ca fiind adrese obisnuite).
Program ilustrativ.
b) (1 punct)
Acelasi lucru cu un macro.

III.23) (puncte - vezi in text) (*)
a) (1.5 puncte)
Functie MIPS ce implementeaza functia C "strcat" (cu retur prin $v0):

```
char *strcat(char *dest, const char *src);
```

care concateneaza o copie a stringului pointat de "src" la sfarsitul stringului pointat de "dest" (suprascriind caracterul nul de la sfarsitul stringului pointat de "dest", a.i. in final sa rezulte un string mai lung) si returneaza "dest" (putem considera "dest", "src" si returul ca fiind adrese obisnuite).
Program ilustrativ.
b) (1 punct)
Acelasi lucru cu un macro.

III.24) (puncte - vezi in text) (*)
a) (3.5 puncte)
Functie MIPS ce implementeaza functia C "strstr" (cu retur prin $v0):

```
char *strstr(const char *s1, const char *s2);
```

care cauta in stringul pointat de "s1" prima aparitie ca substring a stringului pointat de "s2" si returneaza adresa de unde incepe aceasta aparitie sau NULL (0) daca nu exista nici o aparitie (putem considera "s1", "s2" si returul ca fiind adrese obisnuite).

Program ilustrativ.
b) (1 punct)
Acelasi lucru cu un macro.

III.25) (5 puncte)
Functie MIPS ce implementeaza functia C "sscanf" (doar cu formatele "%d" si "%s" si fara alte caractere in stringul de format). Program ilustrativ.

III.26) (20 puncte)
Implementati un alocator de memorie astfel:
- declarati in zona ".data":
 * o zona mare "mem" (.space) din care ulterior se vor aloca diverse bucati;
 * un vector "log" (declarat initial tot cu .space, dar folosit ulterior ca vector de word), in care se vor inregistra adresele bucatilor alocate si lungimile lor;
 * alte variabile necesare gestionarii vectorilor de mai sus (de ex. pt. a retine dimensiunile lor si pana unde s-au ocupat);
- scrieti doua functii:
 "malloc" - primeste ca parametru un word "d", gaseste in "mem" o zona nerezervata de "d" octeti, o rezerva (adaugand la "log" doua worduri - offsetul zonei fata de inceputul vectorului si dimensiunea ei "d"), si returneaza (prin $v0) adresa ei de memorie; daca nu exista o zona libera de dimensiunea ceruta nu rezerva nimic si ret. 0;
 "free" - primeste ca parametru un word, desemnand o adresa de memorie (care se doreste a fi din zona de date statice, unde se afla "mem"), cauta in "log" inregistrarea ce corespunde acestei adrese, o elimina, apoi translateaza celelalte inregistrari (sau o pune pe ultima in locul celei eliminate); daca nu exista o asemenea inregistrare, nu face nimic;
 eventual (pentru inca 10 puncte) "compact" - translateaza zonele alocate in "mem" si modifica inregistrarile corespunzatoare in "log" a.i. zonele alocate sa fie adiacente.
Program ilustrativ.

III.27) (12 puncte)
Folosind metoda salturilor indirecte emulati un interpretor de comenzi de tip "command.com" ca in exemplul de la sectiunea 1h, dar care sa accepte si comenzi cu argumente. Mai exact, comenzile acceptate la intrare si rezultatul scris la iesire in fiecare caz sunt:

"a nr1 nr2 ... nrn" (adunare, n>=1) --> scrie suma nr1+...+nrn
"s nr1 nr2" --> scrie diferenta nr1-nr2
"n nr" --> scrie opusul -nr

in cele de mai sus, "nr", "nri" sunt intregi byte, iar intre componentele comenzilor (nume, nr-uri) sunt caractere blank; fiecare comanda va fi implementata printr-o procedura cu parametri (adunarea are numar variabil de parametri);
vor fi implementate doua comenzi suplimentare, care nu vor putea aparea in fluxul de intrare si care se vor executa automat atunci cand vor fi intrunite anumite conditii: o comanda de eroare, care scrie la iesire un caracter "e", si o comanda de terminare, a carei efect este terminarea programului.
Fluxul de intrare va fi un string ce contine comenzi separate prin caractere <LF> (cod ASCII zecimal 10) iar la sfarsit un caracter <EOF> (cod ASCII zecimal 26); fluxul de iesire va fi un string ce va contine rezultatele scrise de comenzi, separate prin caractere <LF>.
Programul va parcurge liniar fluxul (stringul) de intrare procesand succesiv comenzile; pentru fiecare comanda corecta va executa procedura care o implementeaza si care va scrie la iesire rezultatul comenzii si un <LF>; pentru fiecare comanda eronata va executa automat comanda (procedura) de eroare, care va scrie la iesire un "e"; la intalnirea lui <EOF> va executa automat comanda (procedura) de terminare a programului.
De exemplu daca stringul de intrare contine:

"a 1 2 3<LF>s 30 10<LF>a 100 200<LF>s 20<LF>n 1<LF><EOF>"

atunci stringul de iesire va contine:

"6<LF>20<LF>300<LF>e<LF>ff<LF>"

III.28) (6 puncte daca s-a facut problema III.27 sau 14 puncte daca nu)
Ca la problema III.27, dar comenzile se citesc de la consola interactiv, iar rezultatele se afisaza tot la consola, dupa fiecare comanda (ca la "command.com").

Bibliografie:
~~~~~~~~~~~~~

1."Organizarea si proiectarea calculatoarelor - interfata hardware/software",
John L. Hennessy, David A. Patterson, ed. All, 2002,
anexa A
2. "Programare in limbaj de asamblare"
Gheorghe Musca, ed. Teora, 1997
3. "Limbaje si calculator"
Vlad Bazon, ed. Petrion
4. "Initiere in C++ Programare orientata pe obiecte"
Ionut Muslea, ed. Microinformatica, Cluj-Napoca, 1992

DANIEL DRAGULICI
noiembrie-decembrie, 2006
actualizat: 20 decembrie 2006
~~~~~~~~~~~~~