MEMORIA. DECLARAREA DATELOR. INSTRUCŢIUNI DE TRANSFER

1. Memoria principală

🕮 Memoria principală (Main Memory) este locul în care se păstrează instrucţiunile maşină corespunzătoare unui program şi datele necesare acestuia pentru procesare.

În cazul procesorului MIPS32, numărul locaţiilor de memorie este 2^{32}. Fiecare locaţie de memorie este adresabilă printr-o adresă pe 32 de biţi şi conţine 8 biţi (1 byte). În mod similar, un procesor pe 64 de biţi poate adresa 2^{64} adrese de memorie, folosind 64 de biţi.

Memoria principală a procesorului MIPS32 poate fi considerată asemenea unui tablou de octeţi, fiecare locaţie de păstrare a unui astfel de octet fiind indicată de o secvenţă de 32 de biţi:

0xFFFFFFFF	10001000
0xFFFFFFFE	00010010
0xFFFFFFFD	01010101
0xFFFFFFFC	00110011
0xFFFFFFFB	00110011
…….	……
0x00000003	00100010
0x00000002	00101010
0x00000001	00000100
0x00000000	01101001

Figura 1. Memoria principală

? Întrebări:

1) Câţi GB de memorie pot accesa procesoarele MIPS32 (2^{32} bytes de memorie)?
2) Câţi GB de memorie pot accesa procesoarele MIPS64 (2^{64} bytes de memorie)?

🕮 Ideal, locaţiile de memorie ar trebui să se găsească în RAM. Însă întrucât aceasta conduce la capacităţi destul de ridicate, o parte a memoriei logice se găseşte pe hard disk. Modalitatea de a asigura întregul spaţiu de memorie prin împărţirea ei între RAM şi hard disk poartă denumirea de **memorie virtuală.** Avantajul metodei constă în posibilitatea utilizării întreg spaţiului de memorie, făcând însă un compromis de viteză: accesarea informaţiilor de pe hard disk se realizează mai lent. Pentru estomparea acestei diferenţe, părţi din memoria principală care sunt intens folosite în execuţie se păstrează în **memoria cache,** un tip de memorie accesat mult mai rapid de către procesor. Memoria cache este transparentă pentru program. Din punct de vedere al programelor, există numai

memoria principală. Virtualizarea sau memoria cache nu prezintă interes, aplicaţiile cunoscând doar conceptul de memorie principală.

2. Conţinutul memoriei

📖 Informaţia este păstrată în memorie sub formă binară (0 si 1). Aceasta poate reprezenta instrucţiuni de program, date (întregi, caractere, etc.), etc. Deşi toate aceste informaţii se reprezintă identic din punct de vedere hardware, în memorie există secţiuni separate pentru păstrarea diferitelor tipuri de informaţii. Modul de structurare al memoriei este evidenţiat în subcapitolul următor.

Într-o locaţie de memorie se memorează numai 8 biţi. Dar asta nu înseamnă că nu se pot memora secvenţe de dimensiuni mai mari : intrucţiuni (care ocupă 32 de biţi) sau numere mari, etc. În cazul în care trebuie memorată o secvenţă de biţi de mai mult de 8 biţi, atunci se utilizează mai multe locaţii de memorie, atâtea câte sunt necesare pentru păstrarea întregii secvenţe.

? Întrebări:

1) Câte locaţii de memorie sunt necesare pentru păstrarea unei instrucţiuni?
2) La mutarea unei valori dintr-un registru general în memorie câte locaţii sunt necesare pentru a nu se pierde informaţie?

3. Organizarea memoriei

📖 Memoria este organizată astfel încât să păstreze separat instrucţiunile şi datele.

- **0 x 8000 0000 – 0 x FFFF FFFF: Memorie rezervată pentru Kernel**
  Jumătatea superioară a memoriei (în ordinea adreselor) nu este accesibilă programelor utilizatorilor. Aceasta este rezervată pentru sistemul de operare şi ROM.
- **0x7FFF F000 – 0x7FFF FFFF: Memorie inaccesibilă**
  4KB de memorie între memoria rezervată pentru Kernel şi Segmentul de stivă este inaccesibilă;

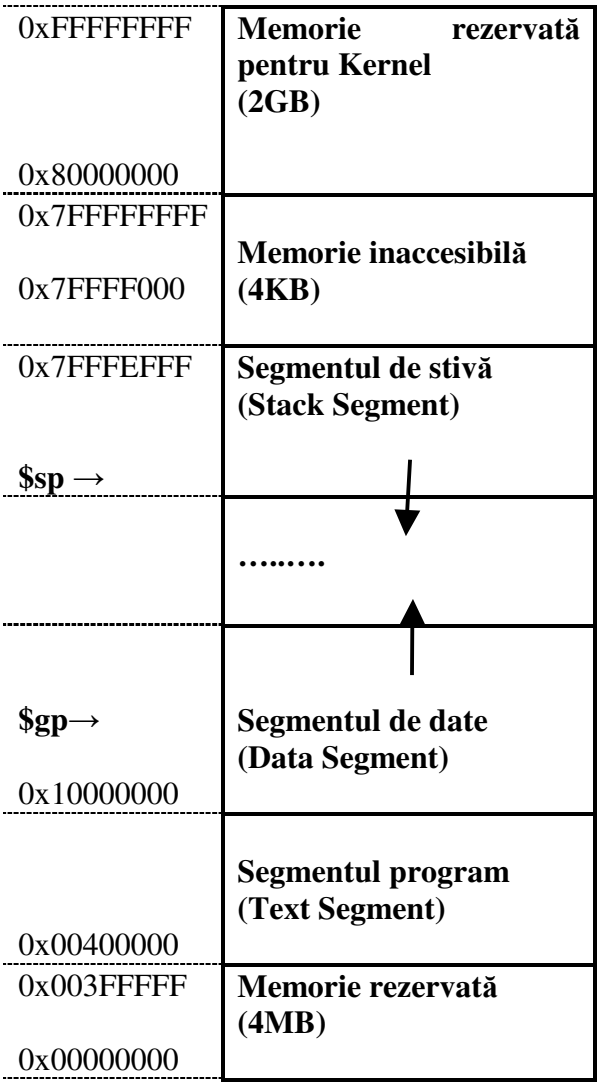


Fig.2. Organizarea memoriei principale

- **… - 0 x 7FFF EFFF: Stack Segment (Segmentul de stivă)**

  Partea de sus a memoriei accesibile este pentru stivă. Variabilele locale, parametrii, etc. sunt introduse/eliminate din stivă prin instrucţiuni de tip push/pop pe măsură ce se apelează o/se iese dintr-o subrutină.

  Stiva nu are alocată o memorie fixă, ci ea ocupă memorie începând de la adresa 0 x 7FFF EFFF. De fiecare dată când necesită o locaţiei de memorie se foloseşte adresa cea mai mare neocupată, adresă care este mai mică decât ultima utilizată de stivă. De aceea se spune că stiva „creşte în jos".

  Adresa de memorie al vârfului stivei este indicată de registrul general **sp** (**S**tack **P**ointer).
- **0 x 1000 0000 - …: Data Segment (Segmentul de date)**

Locaţiile începând cu adresa 0x10000000 sunt utilizate pentru păstrarea datelor necesare programelor. În funcţie de necesităţi, numărul locaţiilor de memorie utilizate în acest scop se extinde în sus. Creşterea „în sus" a segmentului de date, respectiv „în jos" a segmentului de stivă, este posibilă atâta timp cât există locaţii de memorie disponibile.

O parte a datelor pot fi considerate **statice** în sensul că dimensiunea lor nu se modifică în timpul execuţiei programului. Valorile conţinute se pot modifica. Acestea ocupă locaţiile de memorie adresate de adrese mai mici. Deasupra acestora se memorează datele **dinamice**, adică cele care pot fi alocate şi dealocate în timpul programului.

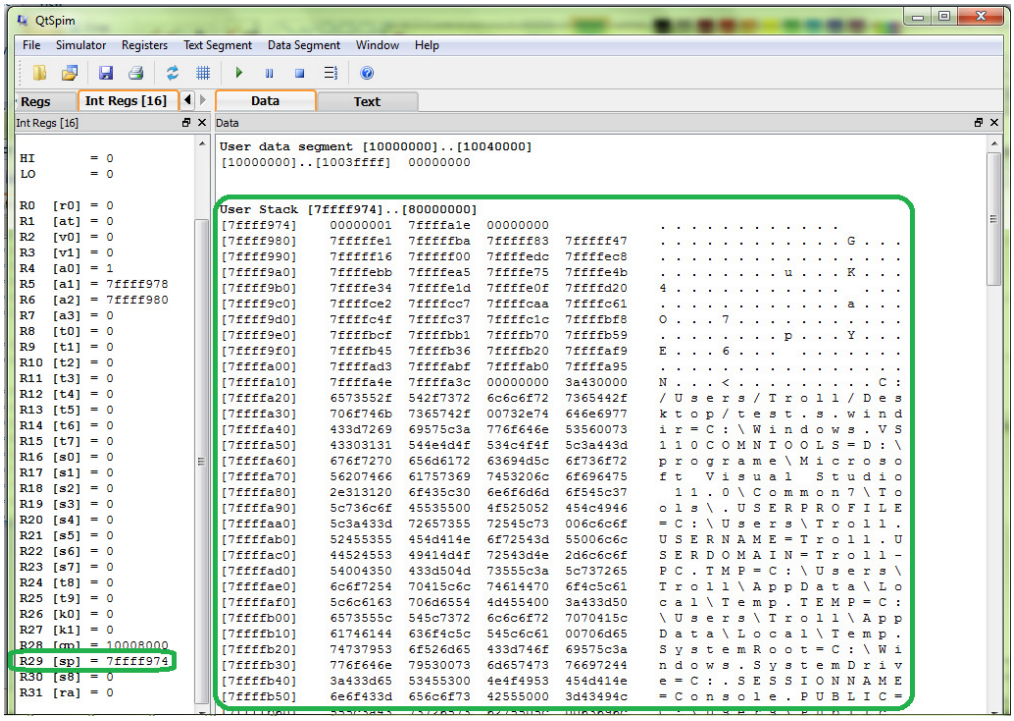
- **0 x 0040 0000 – 0 x 1000 0000: Text Segment (Segmentul Program)**

  Locaţiile de memorie sunt rezervate pentru păstrarea codului maşină al programelor.
- **0 x 0000 0000 – 0 x 003F FFFF: Memorie rezervată**

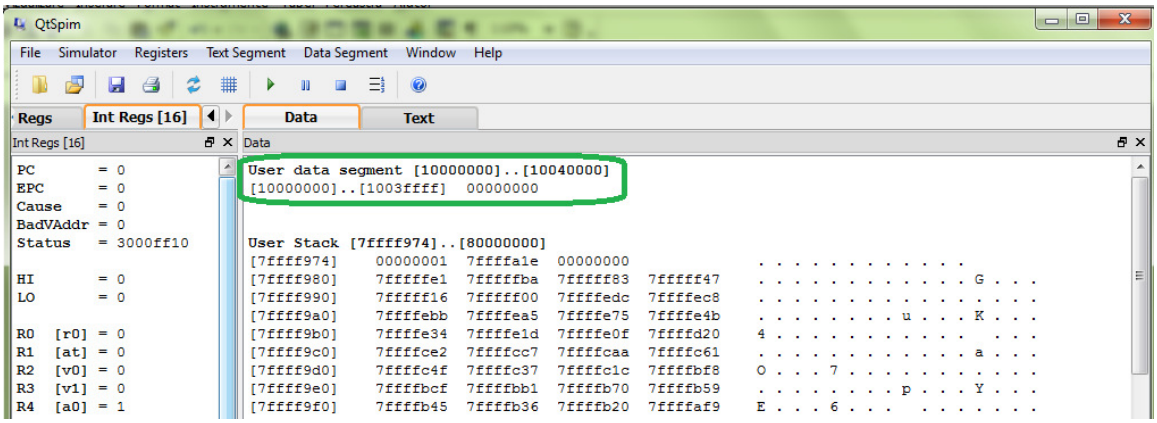
  Primele locaţii de memorie sunt rezervate.

Exerciţiu:

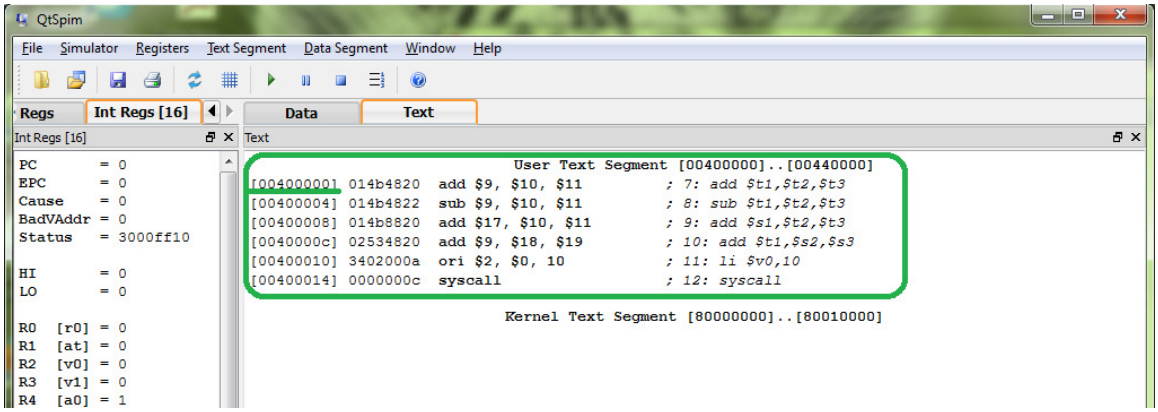
1. Deschideţi QtSpim.
2. Observaţi în panoul DATA al ferestrei principale segmentul de stivă (STACK).



3. Observaţi că valoarea registrului general **sp** este adresa vârfului stivei.
4. Observaţi în panoul DATA al ferestrei principale segmentul de date (DATA):



5. Observaţi că adresa de la care începe segmentul de date este 0x1000 0000;
6. Încărcaţi un program. Observaţi că instrucţiunile maşină sunt încărcate în memorie începând de la adresa 0x0040 0000:



4. Declararea datelor

📖 Datele se declara în cadrul programului imediat după directiva .data. Declararea se realizează după următorul format:

<nume>: <tip> <valori>

unde:

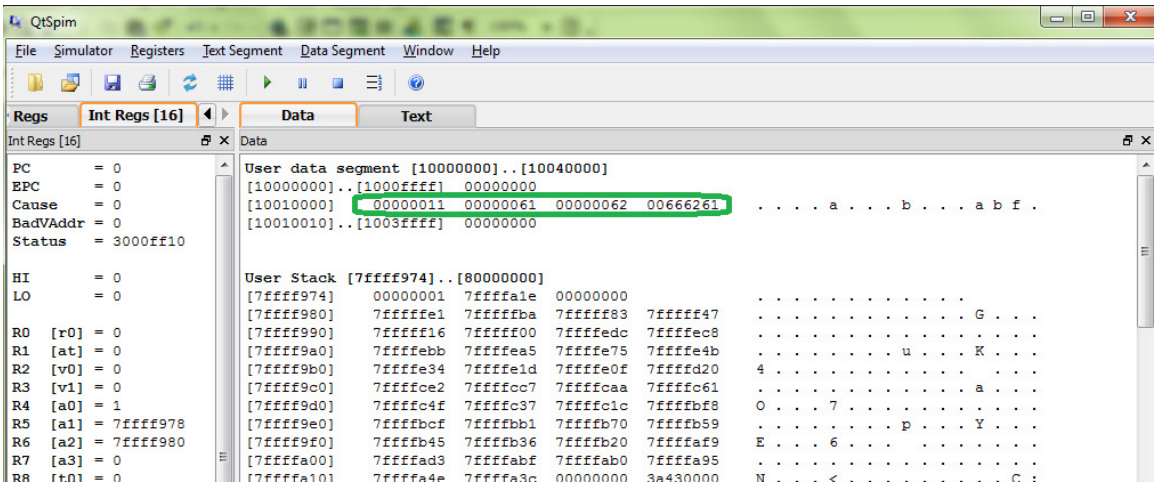
Nume	Reprezintă o etichetă care indică locaţia de memorie alocată la asamblare. Introducerea unei alte valori la respectiva locaţie de memorie conduce la modificarea valorii variabilei. Există posibilitatea declarării de date fără atribuirea unei etichete.
Tip	Reprezintă un tip de date. Poate fi: - **.byte** (byte = 1 octet = 8 biţi) - **.half** (2 octeţi = 16 biţi) - **.word** (word = 4 octeţi = 32 biţi) - **.float** (32 biţi) - numere reale în simplă precizie; - **.double** (64 biţi) - numere reale în dublă precizie; - **.space -** alocă memorie (se utilizează împreună cu **.align n,** care aliniază următoarea valoare pe 2^n octeţi);
Valori	Reprezintă o lista de valori compatibile cu tipul indicat; Valorile se stochează în zona de date, fiecare data ocupând o locaţie corespunzătoare tipului; Valorile se pot scrie numeric zecimal (ex. 2, 10, -1), numeric hexa (ex. 0x21 care înseamnă 33 zecimal), caracter (ex. 'A', 'b'), etc.

💾 **Exerciţiu:**

1. Încărcaţi în QtSpim programul următor:

```
.data
  var1: .word 17
  x: .word'a', 'b'
  y: .byte 'a', 'b'
  z: .byte 102
.text
main:
li $v0,10
syscall
```

2. Observaţi cum se păstrează datele în memorie:
   2.1. De ce apare 0x00000011 pe primii 4 octeţi?
   2.2. De ce apare 0x00000061 pe următorii 4 octeţi?
   2.3. De ce apare 0x00000062 pe următorii 4 octeţi?
   2.4. De ce apare 0x00666261 pe următorii 4 octeţi?



? Întrebări:

1) În ce bază se consideră valoarea 17 asignată variabilei declarate astfel:
   var1: .word 17 ?
2) Ce este o directivă? Ce directive identificaţi în programul încărcat?

5. Metode de adresare

Format	Descriere
(rs)	- adresa o reprezintă conţinutul registrului rs; - codul registrului se păstrează în biţii corespunzători lui rs din instrucţiunea de tip I.
imm	- adresa este dată direct de valoarea imm; - valoarea adresei se păstrează în biţii corespunzători imm din instrucţiunea de tip I.

imm(rs)	- adresa o reprezintă conţinutul registrului rs + valoarea imm; - rs reprezintă adresa de bază, iar imm offsetul; - imm poate fi o valoare pozitivă, negativă sau 0 (când se ajunge în primul caz (rs)).
eticheta	- adresa este cea asociată la compilare etichetei; - se stochează în câmpul imm din formatul I.
eticheta ± imm	- adresa este cea asociată la asamblare etichetei ± valoarea imediată imm; - această adresă se stochează în câmpul imm în formatul I.
eticheta ± imm(rs)	- adresa este cea asociată la compilare + adresa obţinută ca imm(rs);

6. Instrucţiuni de transfer

Operanzii necesari în instrucţiunile aritmetice şi logice trebuie aduşi din memorie în regiştri (**load**). Operaţia inversă, de scriere în memorie este **store**.

Instrucţiunile de transfer sunt în format I:

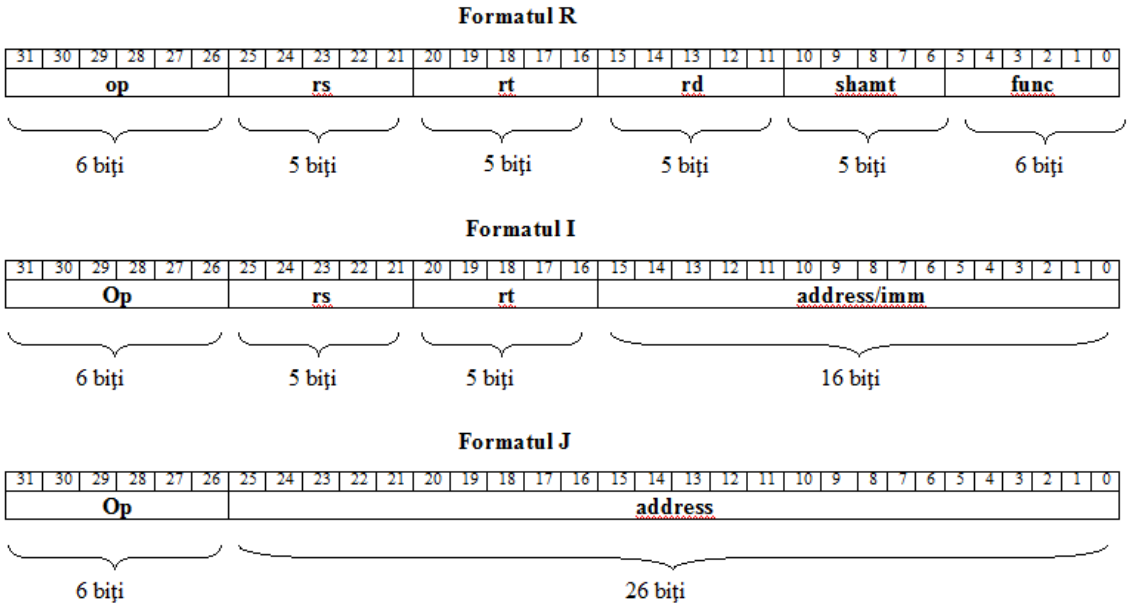


Fig. 3. Formatul instrucţiunilor

📖 Instrucţiunile de transfer sunt prezentate în tabelul următor:

Instrucţiune	**Operaţia efectuată**	**Format**	**Exemplu**
lw rt, imm(rs)	**L**oad **W**ord rt ← memory[rs + imm] În registrul rt se încarcă valoarea păstrată în memorie pe 1 word (4 octeţi) începând cu adresa rs+imm.	I	*lw $t2,3($t1)*
lb rt, imm(rs)	**L**oad **B**yte rt ← memory[rs + imm] Valoarea stocată la adresa rs + imm este considerată cu semn	I	*lb $t2, 3($t1)*
lbu rt, imm(rs)	**L**oad **B**yte **U**nsigned rt ← memory[rs + imm] Valoarea stocată la adresa rs + imm este considerată fără semn	I	*lbu $t2, 0($t1)*
lh rt, imm(rs)	**L**oad **H**alfword rt ← memory[rs + imm] Valoarea stocată pe 16 biţi (2 octeţi) începând cu adresa rs + imm este considerată cu semn;	I	*lh $t2, 3($t1)*
lhu rt, imm(rs)	**L**oad **H**alfword **U**nsigned rt ← memory[rs + imm] Valoarea stocată pe 16 biţi (2 octeţi) începând cu adresa rs + imm este considerată fără semn;	I	*lhu $t2, 3($t1)*
lui rt, imm	**L**oad **U**pper **I**mmediate rt ← imm || 0^{16} Valoarea imediată imm este shiftată cu 16 biţi la stânga şi plasată în rt.	I	*lui $t1, 7*
sw rt, imm(rs)	**S**tore **W**ord memory[rs + imm] ← rt În memorie, pe un word (4 octeţi) începând de	I	*sw $t1, 2($t0)*

	la adresa address se stochează valoarea din registrul rt;		
sb rt, imm(rs)	**S**tore **B**yte memory[rs + imm] ← rt În memorie, pe 1 byte (1 octet) la adresa address se stochează ultimii 8 biţi din rt.	I	*sb $t1,0($t0)*
sh rt, imm(rs)	**S**tore **H**alfword memory[rs + imm] ← rt În memorie, pe 2 bytes (2 octeţi), începând cu adresa address se stochează ultimii 16 biţi menţinuţi în rt.	I	*sh $t1, 2($t0)*

Tabelul 1. Instrucţiuni de transfer

📖 Pseudoinstrucţiunile de transfer sunt prezentate în tabelul următor:

Pseudoinstrucţiune	**Operaţia efectuată**	**Exemplu**
li rt, imm	**L**oad **I**mmediate rt ←imm În registrul rt se încarcă valoarea imm;	*li $t0,1*
la rt, eticheta	**L**oad **A**ddress rt ← address(eticheta) În registrul rt se încarcă adresa asociată variabilei etichetă;	*la $t1,var1*
move rt,rs	**Move** rt ← rs Conţinutul registrului rs este copiat în registrul rt.	*move $t1, $t2*

Tabelul 2. Pseudoinstrucţiuni de transfer

📖 Modurile de adresare utilizate sunt cele prezentate în paragraful precedent.

🕮 Datele se pot scrie sau citi din memorie doar la adrese care sunt multiplii ai dimensiunii tipului respectiv. De exemplu, dacă s-a declarat x de tip word, atunci se va putea scrie în memorie la adresa indicată de x + 4, dar nu şi la adresa indicată de x + 1.

🖫 **Problemă rezolvată:**

Declaraţi valoarea 23 cu eticheta x. Modificaţi valoarea corespunzătoare etichetei x la 5.

```
.data
        x: .word 23
.text
main:
lw $t0,x          # incarca x in registrul t0
li $t1,5          # incarca valoarea 5 in t1
sw $t1,x          # memoreaza valoarea din t1 in x
li $v0,10
syscall
```

🖫 **Probleme propuse:**

1) Introduceţi valorile 1, respectiv 2 în t1, respectiv t2 şi interschimbaţi-le.

2) Scrieţi un program care calculează expresia var3= 8*var1-[var2/16], unde var1=16 şi var2=31.

3) Declaraţi 3 variabile x, y şi z de tip word cu valorile 10, 11, 12.
   a) Încărcaţi în registrul $t1 valoarea word aflată în memorie la adresa lui x + 4, fără a utiliza o altă variabilă în afara lui x. Ce valoare este aceasta?
   b) Stocaţi pe 2 octeţi, la adresa indicată de x + 8 valoarea 14.

4) Fie x eticheta la care se memorează valoarea 5, definită ca word. Să se memoreze la adresa lui x + 4 valoarea polinomului $f = 2x^2 - 4x + 12$.

ⓘ **Mai multe informaţii**

MIPS32™ Architecture For Programmers -Volume II: The MIPS32™ Instruction Set
http://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf

MIPS Instruction Coding
http://www.cs.sunysb.edu/~cse320/MIPS_Instruction_Coding_With_Hex.pdf

MIPS Assembly Language Programmer's Guide
http://www.cs.unibo.it/~solmi/teaching/arch_2002-2003/AssemblyLanguageProgDoc.pdf

Programmed Introduction to MIPS Assembly Language
http://chortle.ccsu.edu/AssemblyTutorial/index.html