Arhitectura sistemelor de calcul – Laboratorul 2

I. Suportul laboratorului 1

1. Registrii MIPS

Denumire simbolica	Numar	Utilizare
$zero	$0	Constanta 0
$at	$1	Rezervat
$v0-$v1	$2-$3	Registri-rezultate si coduri pentru sistem
$a0-$a3	$4-$7	Registri-argument
$t0-$t9	$8-$15, $24-$25	Registri temporari
$s0-$s7	$16-$23	Registri salvati
$k0-$k1	$26-$27	Registri kernel
$gp	$28	Global data pointer
$sp	$29	Stack pointer
$fp	$30	Frame pointer
$ra	$31	Return address

2. Tipurile de date

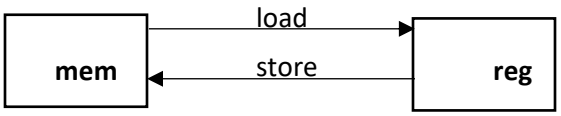
MIPS este un procesor pe 32 de biti (4 bytes), care suporta urmatoarele tipuri principale de date: (toate tipurile au un punct in fata!)

Tipul de date	Descriere
.word	este un tip de date pe 4 bytes care ne ajuta sa stocam valori intregi
.byte	este un tip de date pe 1 byte, pe care il vom utiliza in principiu pentru a stoca valori booleene si caractere
.asciiz	este un tip de date pe care il vom utiliza pentru a stoca siruri de caractere finalizate cu \0
.ascii	este un tip de date care ajuta la stocarea sirurilor de caractere nefinalizate cu \0, nu este recomandat spre a fi utilizat
.space **dim**	este un tip de date care ne ajuta sa definim o zona libera in memorie de dimensiunea **dim** octeti; este utila atunci cand nu stim dinainte valoarea in memorie (ca in majoritatea scenariilor), ci vrem sa o completam in urma executiei unui program.

Declararea de variabila se face cu sintaxa *nume: tip valoare*

Exemple: *x: .word 5*
y: .byte 'a'
str: .asciiz "Mesaj text"
sum: .space 4
ch_res: .space 1

3. Operatiile load si store



Operatie	Forma de utilizare	Descriere
lw (load word)	lw $reg, mem	incarca in registrul $reg valoarea word din memorie; daca exista x declarant in memorie de tipul x:.word 5, atunci il putem incarca in registrul temporar $t0 astfel: lw $t0, x
lb (load byte)	lb $reg, mem	incarca in registrul $reg valoarea byte din memorie; avand y un byte in memorie y: .byte 'a', il incarcam prin lb $t1, y (de exemplu)
la (load address)	la $reg, mem	incarca in registrul $reg adresa din memorie pe care o referim, utilizam in special cand lucram cu siruri de caractere; daca aveam str: .asciiz "Mesaj", il incarcam in $t2 prin la $t2, str
li (load immediate)	li $reg, const	incarca in registrul $reg o valoare constanta, de exemplu li $t3, 4 (am incarcat constanta 4 in registrul $t3)
sw (store word)	sw $reg, mem	salveaza in memorie valoarea din $reg; daca $t0 contine un rezultat pe care vrem sa il salvam in memorie in spatial pe care l-am denumit sum: .space 4, folosim sw $t0, sum (salvam din $t0 in sum)
sb (store byte)	sb $reg, mem	analog sw, dar salveaza un byte in loc de un word; de exemplu, sw $t1, ch_res
move	move $regd, $regs	muta din registrul $regs in registrul $regd (prin copiere); de exemplu move $t0, $v0 inseamna (symbolic) atribuirea $t0 = $v0

4. Instructiuni aritmetice

Operatie	Operanzi	Descriere
abs	des, src	des = abs(src)
add(u)	des, src1, src2	des = src1 + src2
and	des, src1, src2	des = src1 & src2
div(u)	src, reg	Se efectueaza src / reg, se stocheaza in lo catul si in hi restul impartirii mflo $t0 # restul mfhi $t1 # catul
div(u)	des, src1, src2	des = src1 / src2
mul	des, src1, src2	des = src1 * src2
neg(u)	des, src	des = -src
nor	des, src1, src2	des = ~(src1 | src2)
not	des, src	des = ~(src)
or	des, src1, src2	des = src1 | src2

rem(u)	des, src1, src2	des = src1 % src2
sll	des, src1, src2	des = src1 << src2
srl	des, src1, src2	des = src1 >> src2
sub(u)	des, src1, src2	des = src1 – src2
xor	des, src1, src2	des = src1 ^ src2

5. Coduri uzuale pentru apelurile sistem

Cod pentru apelul sistem	Functionalitate	Mod de utilizare
1	PRINT INT	se incarca in $a0 valoarea de afisat; se incarca in $v0 codul 1 (li $v0, 1); syscall
4	PRINT STRING	se incarca in $a0 adresa stringului de afisat; se incarca in $v0, codul 4 (li $v0, 4); syscall
5	READ INT	se incarca in $v0 codul 5 (li $v0, 5); syscall acum, $v0 contine valoarea citita, care se muta in alt registru (in general registru $t), adica se poate face un move $t0, $v0
10	EXIT	se incarca in $v0, codul 10 (li $v0, 10); syscall

6. Instructiuni de comparare

Operatie	Operanzi	Descriere
seq	des, src1, src2	des = 1 daca src1 == src2, altfel des = 0
sne	des, src1, src2	des = 1 daca src1 != src2, altfel des = 0
sge(u)	des, src1, src2	des = 1 daca src1 >= src2, altfel des = 0
sgt(u)	des, src1, src2	des = 1 daca src1 > src2, altfel des = 0
sle(u)	des, src1, src2	des = 1 daca src1 <= src2, altfel des = 0
slt(u)	des, src1, src2	des = 1 daca src1 < src2, altfel 0

7. Salturi conditionate

Operatie	Operanzi	Descriere
beq	src1, src2, et	if src1 == src2 then goto et else continue
bne	src1, src2, et	if src1 != src2 then goto et else continue
bge(u)	src1, src2, et	if src1 >= src2 then goto et else continue
bgt(u)	src1, src2, et	if src1 > src2 then goto et else continue
ble(u)	src1, src2, et	if src1 <= src2 then goto et else continue
blt(u)	src1, src2, et	if src1 < src2 then goto et else continue

8. Saltul neconditionat

Este instructiunea simbolizata prin “j” (jump): j et.

9. Structura unui program MIPS

```
# comentariu
.data
        # zona de declarari de date
.text
        # zona in care vom scrie, de regula, proceduri
main:
        # cod principal
        li $v0, 10 # codul sistem pentru EXIT
        syscall
```

10. Simularea unei structuri repetitive

Vom simula instructiunea for din C:

```
for (i = 0; i < n; i++)
{
        // prelucrare
}
```

astfel:

```
lw $t0, n                       # presupunem ca n este deja cunoscut in memorie
li $t1, 0                       # initializam contorul cu 0, $t1 va fi pe post de i
loop:                           # eticheta de ciclare la care vom reveni
        beq $t1, $t0, exit      # $t1 == $t0 inseamna ca i == n, deci nu se respecta i < n si iesim
        # prelucrare
        addi $t1, $t1, 1        # nu uitam de i++, adica $t1 = $t1 + 1
        j loop                  # revenim la eticheta loop
exit:                           # nu uitam sa implementam eticheta exit
        # cod care urmeaza dupa loop
```

Demo: sa se afiseze toti divizorii unui numar citit de la tastatura.

Putem schita in C/C++ bucata principala de cod care ne intereseaza:

```
for (i = 1; i <= n; ++i)
{
        if (n % i == 0)
                cout << i << " ";
}
```

In MIPS:

```
.data
        n: .space 4                     # inca nu l-am citit si nu ii stiu valoarea, dar stiu ca
                                        # va ocupa 4 bytes in memorie
        sp: .asciiz                     # declaram un spatiu pentru afisare
.text
main:
        li $v0, 5                       # codul apelului sistem corespunzator lui READ INT
        syscall                         # informez sistemul ca vreau sa se uite in $v0
        move $t0, $v0                   # imi mut continutul din $v0 in $t0
        sw $t0, n                       # salvez intregul citit in memorie (nu este obligatoriu
                                        # nici sa declar n in memorie, nici sa il salvez)

        li $t1, 1                       # $t1 este pe post de "i"

loop:
        bgt $t1, $t0, exit              # nu se mai respecta i <= n
        rem $t2, $t0, $t1               # $t2 va fi $t0 % $t1 adica n % i
        beq $t2, $0, afisare            # daca restul este 0, il afisam
continue:                               # marcam o eticheta pentru a sti unde
                                        # sa revenim dupa afisare
        addi $t1, $t1, 1                # $t1 = $t1 + 1, adica i++
        j loop                          # ciclam

afisare:
        move $a0, $t1                   # incarcam in $a0 divizorul, adica pe $t1
        li $v0, 1                       # codul apelului sistem corespunzator lui PRINT INT
        syscall                         # informez sistemul ca vreau sa se uite in $v0

        la $a0, sp                      # incarcam in $a0 adresa stringului de spatiu
        li $v0, 4                       # codul apelului sistem corespunzator lui PRINT STRING
        syscall

        j continue                      # revin in ciclu fix inainte de incrementarea contorului

exit:   li $v0, 10                      # codul apelului sistem corespunzator lui EXIT
        syscall
```

II. Tablouri unidimensionale

Subiectul laboratorului 2 este in jurul tablourilor unidimensionale de date – vectorii.

1. Modul de declarare

Daca stim de dinainte valorile, precizam doar tipul de date si valorile pe care le contine:

```
v: .word 1, 3, 2, 5, 6, 9, 2
```

Daca nu stim de dinainte valorile, precizam cat spatiu trebuie sa ocupe in memorie, relativ la numarul de elemente (calculam numarul de elemente * dimensiunea tipului de date). De exemplu, pentru intregi, tipul de date ocupa 4 bytes, deci daca vrem 10 elemente, avem nevoie de .space 40 (10 elemente * 4 bytes).

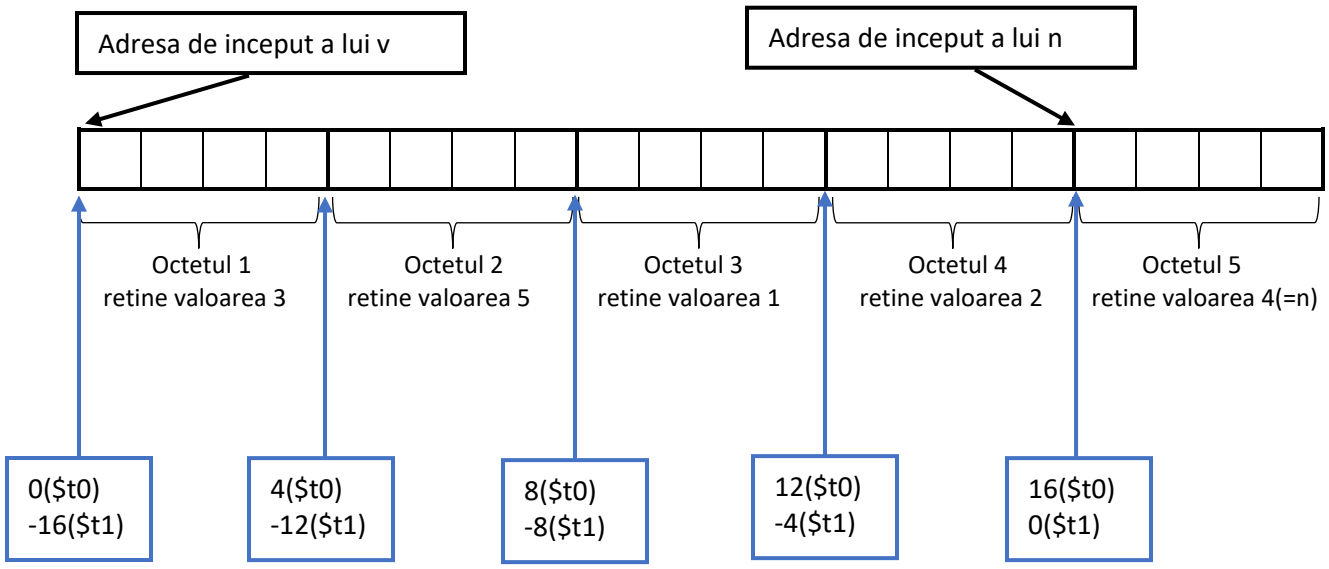
```
w1: .space 40 # vector care stocheaza 10 elemente intregi sau 40 bytes
w2: .space 10 # vector care stocheaza 10 bytes
              # (nu stocheaza intregi pentru ca 10 nu este multiplu de 4)
```

2. Modul de accesare

Memoria este reprezentata liniar, si consideram urmatorul exemplu:

```
v: .word 3, 5, 1, 2      # v este un vector de intregi
n: .word 4               # n este 4, lungimea vectorului
```

Daca incarcam adresele *la $t0, v* si *la $t1, n*, atunci:



Urmarind schema, observam ca putem accesa al treilea element din vector, de exemplu, cu instructiunea *lw $t2, 8($t0)* (primul element este la adresa 0, apoi cresc adresele din 4 in 4).

Problema: varianta de mai sus nu ne permite sa parcurgem elementele vectorului intr-o structura repetitiva, deoarece varianta de adresare are forma **const($reg)** si **nu $reg($reg)**, adica nu pot sa ma folosesc de un registru care creste cu 4 la fiecare iteratie pentru a accesa **$reg_care_creste($reg_adresa)**.

Solutie: Mai exista o varianta de adresare, de forma **adresa($reg)**, care ne permite sa crestem valoarea cu 4 la fiecare iteratie a valorii din $reg. Atunci, daca **v** este vectorul nostru, iar **$t0** contorul care creste, putem accesa elementele cu **v($t0)**.

Demo: sa se afiseze valorile unui vector pe ecran.

```
.data
        v: .word 3, 4, 1, 2, 5
        n: .word 5
        sp: .asciiz " "
.text
main:
        lw $t0, n               # in $t0 retinem dimensiunea
        li $t1, 0               # $t1 este contorul obisnuit
        li $t2, 0               # $t2 creste din 4 in 4
loop:
        bge $t1, $t0, exit      # conditia de iesire cand i >= n

        lw $a0, v($t2)          # incarcam in $a0 elementul curent
                                # aici v($t2) e similar cu v[i] din C
        li $v0, 1               # codul de PRINT INT
        syscall

        la $a0, sp              # incarcam in $a0 adresa spatiului de afisat
        li $v0, 4               # codul de PRINT STRING
        syscall

        addi $t2, $t2, 4        # $t2 creste din 4 in 4 pentru a accesa memoria
        addi $t1, $t1, 1        # $t1 se incrementeaza standard
        j loop                  # revenim la loop

exit:
        li $v0, 10              # codul de EXIT
        syscall
```

Exercitiu propus: sa se determine elementul maxim si numarul de aparitii al acestuia intr-un vector dat in memorie. De exemplu, pentru

```
        v: .word 15, 239, 34, 998, 342, 998, 324, 998, 24, 13
        n: .word 10
```

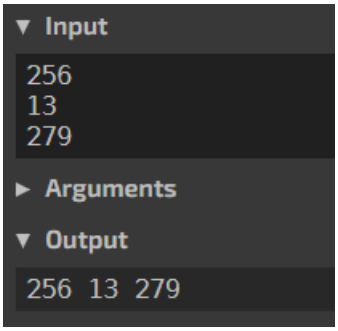
se va afisa pe ecran *Maximul este 998 cu numarul de aparitii 3*

3. Citirea elementelor intr-un vector

Daca registrul **$t0** contine valoarea citita de la tastatura, atunci se poate salva in vector cu **sw $t0, v($t1)** unde **$t1** creste din 4 in 4. Urmatorul program arata modul in care se citesc elementele vectorului de la tastatura si afisarea lor pe ecran:

```
.data
        v: .space 12            # pot retine 3 intregi sau 12 variabile byte
        n: .word 3              # n = 3, deci lucrez cu 3 intregi
        sp: .asciiz " "
.text
main:
        lw $t0, n               # $t0 = n
        li $t1, 0               # $t1 este contorul obisnuit
        li $t2, 0               # $t2 creste din 4 in 4

loop_read:
        bge $t1, $t0, afisare   # daca i >= n se merge la afisarea vectorului

        li $v0, 5               # READ INT
        syscall
        move $t3, $v0           # obtinem elementul in $t3
        sw $t3, v($t2)          # il salvam in vector

        addi $t1, $t1, 1        # incrementam contorul
        addi $t2, $t2, 4        # crestem cu 4 indexul care acceseaza memoria
        j loop_read

afisare:
        li $t1, 0               # reinitializam $t1
        li $t2, 0               # reinitializam $t2
loop_write:
        bge $t1, $t0, exit      # daca i >= n se merge la exit
        lw $a0, v($t2)          # incarcam in $a0 elementul curent
        li $v0, 1               # PRINT INT
        syscall

        la $a0, sp              # se incarca in $a0 spatiul
        li $v0, 4               # PRINT STRING
        syscall

        addi $t1, $t1, 1        # incrementam contorul
        addi $t2, $t2, 4        # crestem cu 4 indexul care acceseaza memoria
        j loop_write

exit:
        li $v0, 10
        syscall
```

Observatie: pentru a testa programul, introduceti valorile cate una pe linie. Exemplu:



Daca nu cunoastem de la inceput numarul de elemente din vector, putem sa ii alocam spatiu mai mult (mai mult decat consideram ca ar fi suficient) si sa citim n-ul de la tastatura. In general, declaram .space 400 (adica ceea ce ii corespunde lui v[100]).

Exercitiu: se citesc de la tastatura n (n > 0) si un vector de intregi de dimensiune n. Sa se verifice daca elementele sunt in ordine crescatoare. In caz afirmativ, sa se afiseze pe ecran *Elementele sunt in ordine crescatoare*, in caz contrar sa se afiseze *Elementele nu sunt in ordine crescatoare; primul index gresit este 2* daca incepand de la elementul de pe pozitia 2 (numerotarea este de la 0) apar greseli de ordonare.

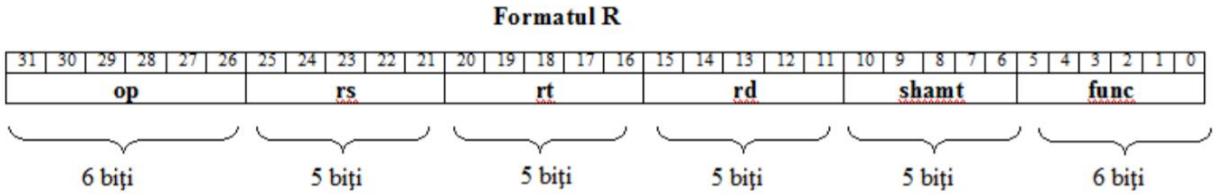
Pentru vectorul *v: .word 3, 4, 1, 2, 3* s-ar afisa mesajul conform caruia de la indexul 2 valorile nu mai sunt in ordine crescatoare.

III. Formatul intern R

Instructiunile pe care le utilizam in cadrul procesorului MIPS sunt reprezentate intern folosind 3 standarde, si anume formatele interne R, I si J. Operatii reprezentative sunt:

- Pentru formatul R: add, sub, and, or, sll etc.
- Pentru formatul I: lw, sw, beq etc.
- Pentru formatul J: j.

Instructiunile sunt stocate pe 32 de biti (4 bytes), iar registrii se reprezinta in instructiuni pe 5 biti. Formatul intern R are urmatoarea reprezentare:



unde:

op = operatia de baza care, in cazul intructiunilor in format R, este mereu 000000;
rs = registru sursa – registrul care contine primul argument (numarul lui, vezi primul tabel);
rt = registru sursa – registrul care contine al doilea argument (din nou, numarul lui);
rd = registru destinatie – registrul in care se stocheaza rezultatul obtinut in urma operatiei;
shamt = shift amount – folosit la operatiile care utilizeaza shiftare. (sll, slr)
func = functie – combinata cu op indica operatia care se aplica.

Valorile pentru func sunt date, nu trebuie invatate!

Exemplu: sa se reprezinte scrierea interna pentru operatia add $t0, $t1, $t2. **(= add $8, $9, $10)**

Componenta	**Valoare**	**Reprezentare interna**
op	0 pe 6 biti	000000
rs	$t1 = $9	01001 (9 in binar pe 5 biti)
rt	$t1 = $10	01010 (10 in binar pe 5 biti)
rd	$t0 = $8	01000 (8 in binar pe 5 biti)
shamt	0 pe 5 biti (nu se face shiftare)	00000
func	100000 (**este dat**, e functia de adunare)	100000

Atunci, instructiunea add $t0, $t1, $t2 se reprezinta in binar ca

000000 01001 01010 01000 00000 100000

Prelucram sirul in grupe de cate 4

0000 0001 0010 1010 0100 0000 0010 0000 (instructiunea cod masina binar)

Si transcriem in hexa: 0x012A4020 (instructiunea cod masina hexa)

Exercitiu: sa se calculeze instructiunea cod masina binar si hexa pentru operatia *sll $t0, $t3, 2*. Observatii utile:

- op este 000000 pentru ca formatul este R;
- func este 000000 in cazul sll (shift left logical);
- se face shiftare cu 2, deci se completeaza si campul shamt;
- $t0 este registrul destinatie, $t3 este registrul sursa **rt**, dar nu avem registrul sursa **rs**, asa ca va fi completat cu **00000**.

Exercitiu: sa se calculeze instructiunea cod masina binar si hexa pentru operatia *sub $s0, $s0, $s1* stiind ca func este 100010.