

Tutoriat 2 - Arhitectura sistemelor de calcul

Stan Bianca-Mihaela, Stancioiu Silviu

November 2020

**ORICINE: ESTE IMPOZIBIL SA FACI SI MATERIA DE
SEMINAR SI CEA DE LABORATOR INTR-UN SINGUR
TUTORIAT**

TUTORII DE ASCI:



1 Reprezentarea numerelor în calculator

1.1 Noțiuni generale

Într-un calculator nu putem reprezenta numerele aşa cum o facem în matematică (sau cel puțin nu o putem face într-un mod performant), aşa că există câteva convenții legate de reprezentarea numerelor în calculator. În calculator numerele sunt reprezentate binar folosind un număr finit de biți (un bit poate 1 sau 0). De regulă se folosește un multiplu de 8 biți pentru a reprezenta un număr.

Considerăm următoarele operații hardware la nivel de bit:

a	b	$a b$ (or)	$a \& b$ (and)	$a \wedge b$ (xor)	a	$\sim a$ (not)
0	0	0	0	0	0	1
0	1	1	0	1	1	0
1	0	1	0	1		
1	1	1	1	0		

Pe un număr reprezentat în binar pe un număr FINIT de biți putem aplica operații logice pe biți: *or*, *and*, *xor*, etc. Ele se efectuează apilcând bit cu bit operațiile logice respective, conform tabelului anterior.

Exemplu pentru operația de *and* logic pentru două numere reprezentate pe 8 biți:

$$\begin{array}{r} 11000101 \\ \textcircled{\text{\textampersand}} \\ 01000110 \\ \hline 01000100 \end{array}$$

Pe lângă operațiile logice pe biți, putem aplica și operațiile aritmetice binare (cele discutate în tutoriatul anterior, adică adunare, scadere, etc) asupra unui număr reprezentat binar pe un număr finit de biți.

În cazul adunării, dacă trebuie să facem transport pe o poziție mai mare decât numărul de biți pe care îl avem la dispoziție, acel transport se pierde. Iar la scădere, dacă e nevoie să ne împrumutăm de la o poziție mai mare decât numărul de biți pe care îl avem la dispoziție, atunci împrumutul se face "automat".

Exemplu pentru adunarea și scăderea a două numere reprezentate pe 8 biți:

$$\begin{array}{r} 11000101 \\ + 01000110 \\ \hline 00001011 \end{array} \quad \begin{array}{r} 00001011 \\ - 01000110 \\ \hline 11000101 \end{array}$$

Putem aplica și operații unare asupra acestor numere (complement față de 1 și complement față de 2). Complementul față de 1 înseamnă ca luăm fiecare bit din reprezentarea numărului și îl negăm (adică dacă e 1 îl transformăm în 0, iar dacă e 0 îl transformăm în 1).

Complementul față de 2 înseamnă ca prima oară calculăm complementul față de 1 al numărului, apoi adunăm 1 aritmetic (adunare ca în exemplele anterioare).

Exemple pentru un număr reprezentat pe 8 biți:

$$\begin{array}{r} \square \quad 01001100 \\ \hline 10110011 \end{array} \qquad \begin{array}{r} \square \quad 01001100 \\ \hline 10110100 \end{array}$$

Observăm că pentru a scădea două numere, putem face suma dintre primul număr și complementul față de 2 al celui de-al doilea număr (TODO: luați un exemplu și verificați că e adevărat). De aceea, deși sunt diferite instrucțiunile pentru adunare și scădere din limbajul de asamblare, intern sunt realizate de același circuit, numit sumator.

1.2 Reprezentarea numerelor naturale ca întregi fără semn

În limbajul C, această reprezentare este folosită în cazul tipuilor de date întregi însoțite de "unsigned": unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long

Fie un tip de numere naturale pe n biți, atunci acel tip de date poate stoca toate numerele naturale cuprinse între 0 și 2^{n-1} inclusiv.

Exemplu pe 8 biți: cea mai mică valoare care poate fi stocată pe 8 biți folosind această reprezentare este 0 (adică 00000000 intern), iar cea mai mare valoare care poate fi reprezentată este 255 (adică 11111111 intern, practic este cel mai mare număr care poate fi reprezentat cu 8 biți). După cum am discutat mai devreme (faptul că transportul se pierde după ultimul bit), rezultă că operațiile cu aceste numere se fac modulo 2^n .

Exemplu: dacă avem numărul 255 (ambele stocate ca numere naturale fără semn pe 8 biți, $[255]_8 = \overline{11111111}$) și adunăm la el $[1]_8 = \overline{00000001}$, operația internă care se execută este: $11111111 + 00000001$. Se va transportă un bit până la ultimul bit, unde se pierde, deci ne va rămâne 00000000. Adică fix 0. 0 este egal cu 256 mod 256 (pentru că $256 = 2^8$).

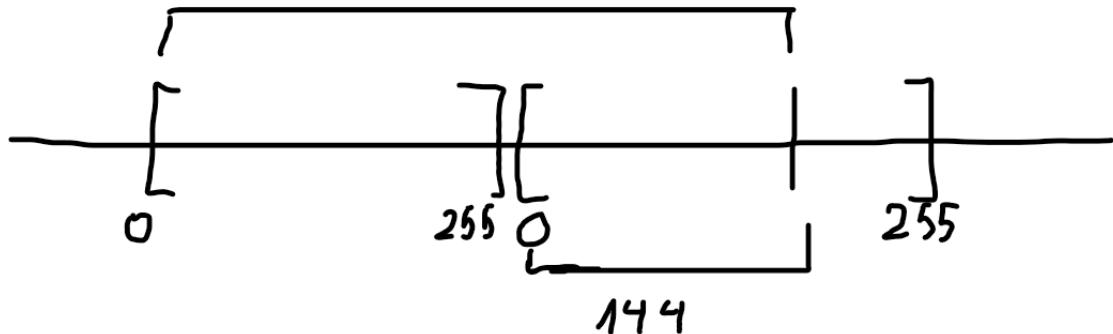
Alt exemplu: tot pe 8 biți lucrăm și avem de adunat 128 cu 129 ($[128]_8 = \overline{10000000}$) și adunăm la el $[129]_8 = \overline{10000001}$, adică $10000000 + 10000001$. Folosind regulile de calcul de mai sus ne va da 00000001, adică 1. $1 = (128 + 129) \text{ mod } 256 = 257 \text{ mod } 256$.

Observație:

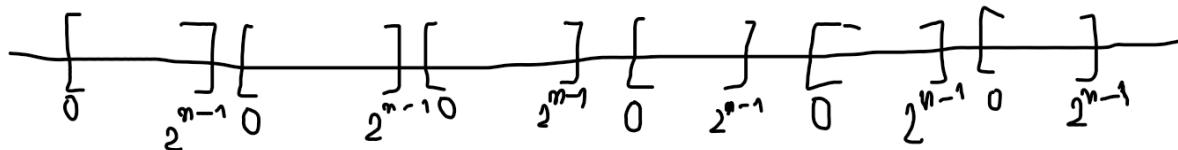
Atunci când suma a două numere reprezentate în acest mod depășește valoarea maximă suportată de tipul de date, rezultatul va fi strict mai mic decât cel puțin unul din termenii adunării. Datorită acestei proprietăți putem verifica dacă s-a produs overflow în timpul adunării.

Alt exemplu: tot pe 8 biți avem $x = 200$ și $y = 200$. Dacă facem $z = x + y$, vom avea că z are valoarea 144. Să vedem și în binar ce se întâmplă: avem $x, y = 11001000$, deci $x + y = 11001000 + 11001000 = 10010000$ deoarece s-a pierdut transportul. Putem să și vizualizăm acest exemplu:

400



Practic pentru orice operație ne putem imagina că avem o axă imaginară cu intervale de valori cuprinse între 0 și $2^n - 1$. Fiecare operație ne ducă mai în stânga sau în dreapta axei. Punctul în care ajungem este rezultatul calcului, evident, o valoare între 0 și $2^n - 1$ inclusiv.



Secvența de cod: $x = 255; ++x$; face ca x să devină 0 (cum am văzut și într-un exemplu anterior)

Secvența de cod: $x = 0; -x$; face ca x să devină 255

Secvența de cod: for ($x = 0; x < 256; ++x$) este un ciclu infinit deoarece x mereu va fi mai mic decât 256 (256 nu poate fi reprezentat pe 8 biți folosind această reprezentare), deci condiția $x \neq 256$ este mereu adevărată. Practic se va cicla la infinit printre valorile 0, 1, ... 255.

De notat că la efectuarea testului $x < 256$ operanzii sunt convertiți la tipul *int*, iar comparația se face în cadrul tipului *int*.

1.3 Reprezentarea numerelor întregi în complement față de 2

În limbajul C, reprezentarea în complement față de 2 se folosește în cazul tipurilor întregi ne-însotite de 'unsigned':

char, signed char, short, signed short, int, signed int, long, signed long, long long, signed long long. Un tip de date cu semn pe n biți poate ține numere întregi cu valori cuprinse între -2^{n-1} și $2^{n-1} - 1$. Dacă avem un număr cuprins în acest interval și vrem să-l reprezentăm folosind această reprezentare, avem două variante:

1. Dacă numărul este pozitiv, atunci pur și simplu scriem reprezentarea lui binară.
2. Dacă numărul este negativ scriem complementul său față de 2.

Exemplu: În limbajul C, pentru tipul char avem: $n = 8$.

El poate ține valori întregi cuprinse între -128 și 127. Modul de reprezentare a câtorva valori:

127: 01111111

126: 01111110

1: 00000001

0: 00000000

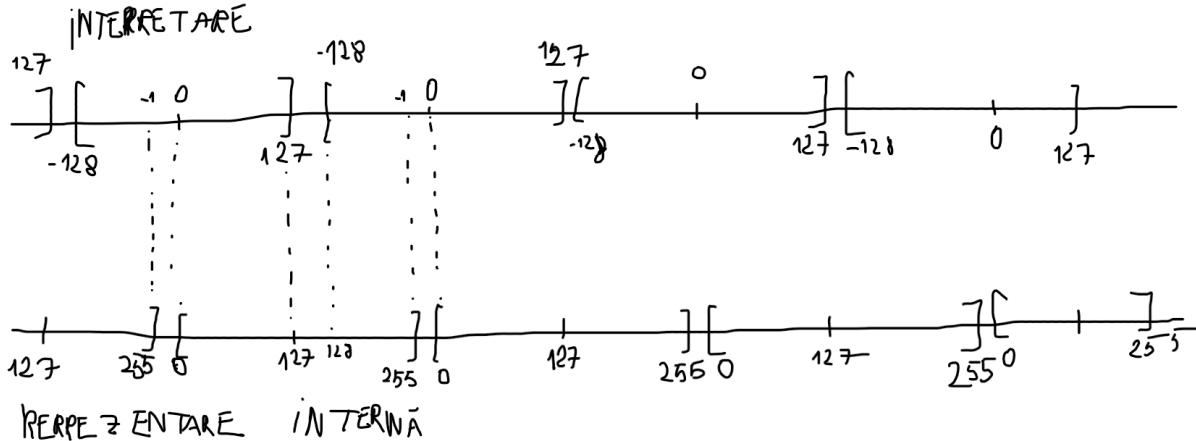
-1: 11111111
-2: 11111110

-127: 10000001
-128: 10000000

Observație:

Numerele negative au valoarea celui mai semnificativ bit egală cu 1, pe când numerele pozitive au valoarea aceluia bit egală cu 0.

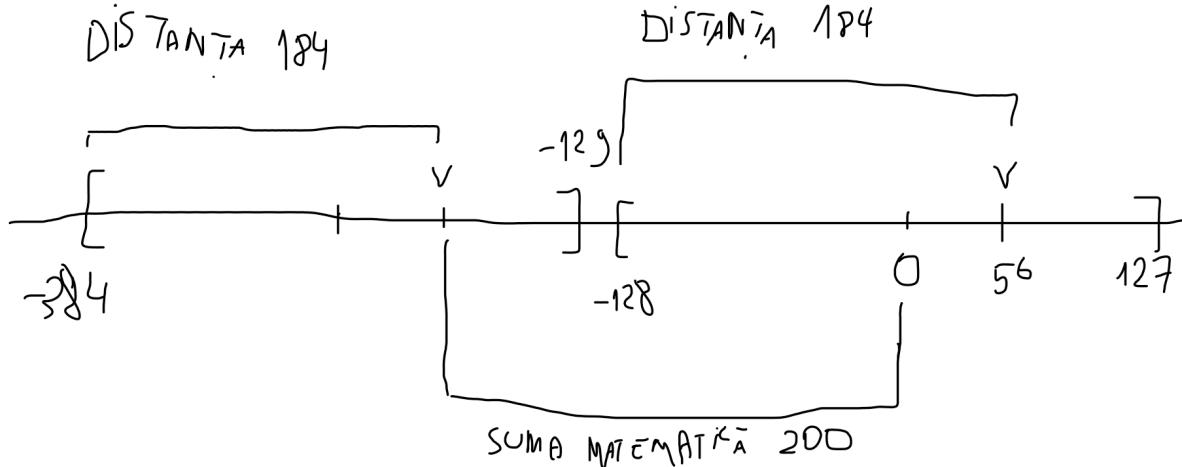
Asemănător cu reprezentarea intuitivă de la numerele unsigned, aici ne putem imagina două axe infinite de "intervale cu valori întregi". Una cu valori cuprinse între -2^{n-1} și $2^{n-1} - 1$, iar cealaltă cu valori cuprinse între 0 și $2^n - 1$. Practic avem o bijectie astfel: pe axa cu numere negative numerele între 0 și $2^{n-1} - 1$ sunt mapate tot pe același valori de pe cealaltă axă, iar numerele negative sunt mapate pe "intervalul" dintre 2^{n-1} și $2^n - 1$ pe cealaltă axă. Prima axă este cea a interpretării (adică valoarea pe care o interpretăm noi că o are numărul), iar cea de a doua axă este reprezentarea lui în calculator. (adică dacă transformăm numerele de pe cea de a doua axă în baza 2, obținem reprezentarea lor internă în calculator). Pentru 8 biți avem:



Operațiile se fac tot 2^n , dar translatat (deplasamentul se măsoară față de începutul intervalului $\{-2^{n-1}, \dots, 2^{n-1} - 1\}$).

Exemplu: În limbajul C, avem:

```
char x, y, z; /* n = 8, M = {-128, ..., 127} */  
x = -100; y = -100; z = x + y;  
printf("%d", (int)z); /* afișează 56 */
```



Exemplu: Fie $x = 32$, $y = 41$. Calculați $z = x - y$ folosind reprezentarea în complement față de 2 pe 8 biți.

Avem $n = 8$, deci mulțimea valorilor pe care le putem stoca este $M = \{-128, \dots, 127\}$.

Deoarece x și y sunt pozitive și aparțin mulțimii de mai sus, le vom reprezenta direct în baza 2. Deci $x = 00100000$ și $y = 00101001$.

Calculăm complementul lui y față de 2, pentru asta va trebui ca prima oară să calculăm complementul față de 1, deci vom avea: 11010110. Acum adunăm 1 și avem 11010111. Tocmai am obținut complementul lui față de 2. Acum trebuie doar să adunăm x cu rezultatul obținut, avem: $00100000 + 11010111 = 11110111$. Acest număr înseamnă 247. Deoarece cel mai semnificativ bit este 1, înseamnă că avem un rezultat negativ, deci pentru a afla valoarea rezultată îl scoatem pe z din următoarea ecuație: $256 + z = 247$. În acest caz ne va da că $z = -9$. Deci calculul pentru situații de genul este $2^n + z = \text{valoare_interpretata_din_binar}$. O intuție în acest caz ar fi să ne uităm pe diagramă și să ne gândim că 256 înseamnă 0 (și chiar asa și înseamnă după bijecția stabilită de noi). Ne gândim care este diferența între origine și rezultatul nostru, iar această diferență este chiar diferența din diagrama interpretării.

Alt mod de a interpreta acest exercițiu este să ne imaginăm că prelungim intervalul pe axa noastră infinită, facem calculul matematic, calculăm diferența dintre rezultatul nostru și începutul intervalului în care pică rezultatul, iar apoi să adunăm această diferență la capătul de început al intervalului nostru. (exemplul anterior luat din cursul lui Drăgulici). În cazul acesta nu este nevoie de aşa ceva pentru că oricum -9 se află în range-ul în care acceptăm valori.

Dacă aveți la examen un exercițiu asemănător, urmați metoda de rezolvare cu $2^n + z = \text{valoare_interpretata_din_binar}$, nu a doua metodă de rezolvare pe care v-am prezentat-o.

Exemplu: În limbajul C considerăm următoarea secvență de cod:

```
char x; /* p = 8 */
short y; /* q = 16 */
x = -127; /* se reprezintă -127 ca întreg cu semn pe 8 biți */
y = x; /* se extinde reprezentarea din x de la 8 biți la 16 biți, prin propagarea bitului de semn*/
```

Verificați că valoarea reprezentată în y este tot -127 .

Avem: $[-127]_8^s = (256 - 127)_2^8$ (deoarece $-127 < 0$) $= (129)_2^8 = 10000001$. (Acestea sunt notațiile lui Drăgulici

pentru treaba asta, vom discuta mai multe pe tema asta când ne vom apuca de rezolvat modele de examen.). Prin propagarea bitului de semn la 16 biți, obținem: $1111111100000001 = ((2^{16} - 1) - (2^7 - 1) + 1)_2^{16} = (65535 - 127 + 1)_2^{16} = (65409)_2^{16}$. Aceasta este reprezentarea ca întreg cu semn a unui număr $z < 0$, deoarece bitul 15 (de semn) este 1.



Atunci $[z]_{16}^s = (65536+z)_2^{16} = (65409)_2^{16}$, deci $65536 + z = 65409$, deci $z = -127$.

Exercitiul 1 : [RESTANTA SEPT 2020] Calculati $z=x-y$, ($x=112$, $y=37$) folosind reprezentarea in complement fata de 2 pe 8 biti. Se vor explicita reprezentarile lui x si y , complementul fata de 1 si cel fata de 2 al reprezentarii lui y , obtinerea reprezentarii lui z , interpretarea acesteia ca numar in baza 10.

$$[x]_8 = [112]_8 = 01110000$$

$$[y]_8^s = [37]_8 = \overline{00100101}$$

complementul fata de 1 al lui y este: $\overline{11011010}$

adaugam 1 la complementul fata de 1 si obtinem complementul fata de 2: $1 + \overline{11011010} = \overline{11011011}$

$$\Rightarrow [z]_8^s = \overline{01001011} = [2^6 + 2^4 + 2^3 + 2^1 + 2^0]_8 = [75]_8$$

\Rightarrow reprezentarea lui z ca numar in baza 10 este: 75

Exercitiul 2 : Calculati $z=x-y$ ($x=12$, $y=67$) folosind reprezentarea in complement fata de 2 pe 8 biti.

2 MIPS (Ne pare rău seria 15, nu știm x86 assembly, chiar am vrea să știm și să va putem ajuta să înțelegeți)



2.1 Notiuni generale

Limbajul de programare MIPS assembly este modul prin care programăm un procesor care are arhitectura MIPS. Această arhitectură este una de tip RISC (Reduced Instruction Set Computer), spre deosebire de arhitectura x86 care este de tip CISC (Complex Instruction Set Computer). Procesoarele de tip MIPS sunt mici, simple și consumă puțin curent, de aceea acestea se găsesc de regulă pe dispozitive precum: teste de sarcină, frigidere, mașini de spălat, etc. Procesoarele MIPS sunt procesoare pe 32 de biți (4 bytes).

Intr-adevăr, probabil vor fi puține situații (most likely nu vor fi deloc) în care veți coda în MIPS în cariera voastră de programatori, dar înțelegerea acestui limbaj vă va prinde foarte bine pe viitor. MIPS este un limbaj de asamblare foarte ușor (comparativ cu x86 sau x64), iar dacă îl veți învăța vă va fi usor după să treceți la limbaje de asamblare precum x86 sau x64 assembly (pe acestea trebuie să le știți dacă vreți să lucrați la antiviruși, să analizați alte programe, să modătați jocuri, etc).

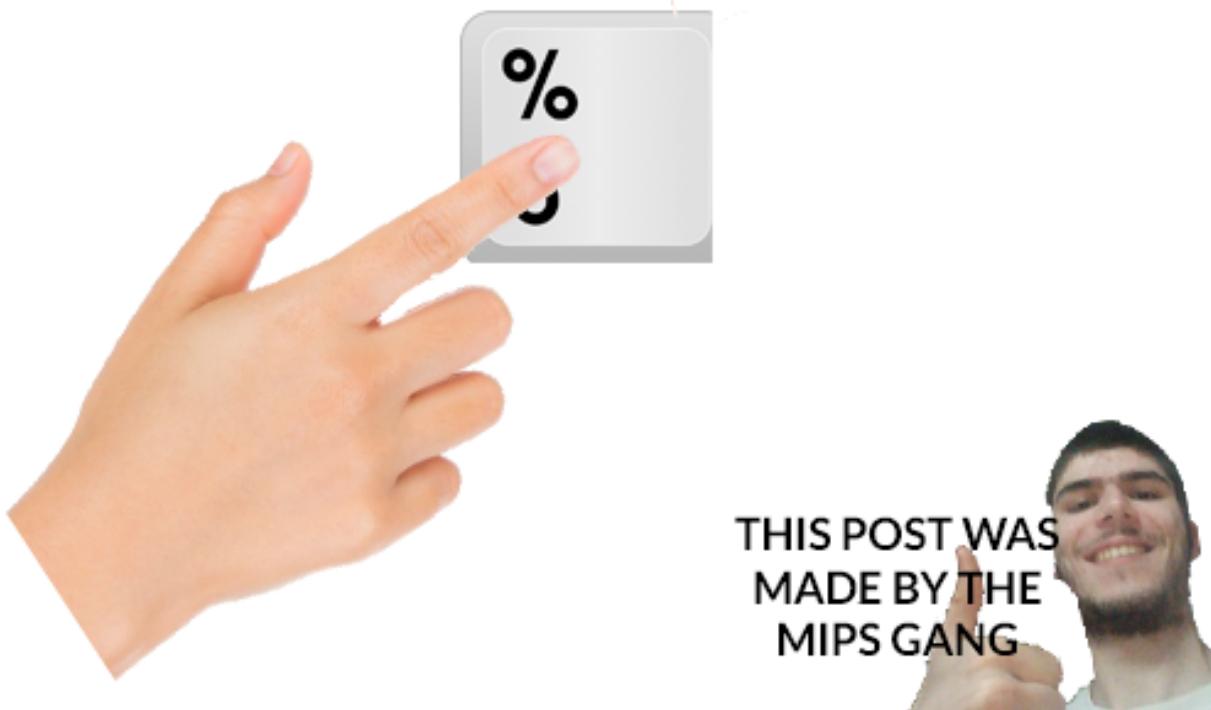
2.2 Lucrul cu registrii

În limbaje de programare high level precum C/C++ sau Python suntem obișnuiți să lucrăm cu variabile pentru a realiza operații/ a stoca date. În limbajele de asamblare, lucrul cu memoria nu este atât de "direct". Aici pentru a lucra cu memorie, această memorie trebuie mai întâi copiată în registrii de pe procesor, apoi se fac operațiile folosind registrii de pe procesor, iar la final se copiază înapoi în memoria RAM rezultatele. (cam asta ar fi explicația la lucrul cu registrii pe scurt, vom vedea în cîteva exemple mai jos cum se lucrează)

Regiștri MIPS

Când scriem cod în MIPS, prefixăm cu \$ fiecare registru pe care îl folosim (în x86 at&t assembly prefixăm cu %).

IMAGINE PREFIXING YOUR REGISTERS WITH % INSTEAD OF \$



În MIPS avem următorii registri:

- \$zero
- \$at
- \$a0-\$a4
- \$v0-\$v1 (în \$v0 vom pune codul pentru apelurile de sistem)
- \$t0-\$t7, \$t8, \$t9 (sunt registri temporari, de regulă cu ei vom lucra în main)
- \$s0-\$s8 (registri salvați, de regulă vom lucra cu ei în interiorul procedurilor)
- \$sp, \$fp (registri pentru controlul stivei)

- \$ra

Tipuri de date MIPS

Acstea sunt folosite pentru a declara date "stative" la începutul programului. Ne putem imagina că ele sunt stocate direct in RAM, iar noi trebuie să le copiem în registri pentru a lucra cu ele, iar apoi din registri să introarcem înapoi valorile în "memoria RAM". Tipurile de date sunt prefixate cu caracterul punct '.' în MIPS.

Tipurile:

- .word - tip de date pe 32 de biți (4 bytes), folosit pentru a stoca întregi
- .byte - tip de date pe un byte (8 biți) folosit pentru a stoca caractere sau valori booleene
- .ascii
- .asciiz - folosit pentru a ține siruri de caractere. Pe acesta îl vom folosi, nu pe .ascii
- .space *dim* - declară o zonă liberă de memorie de dimensiunea *dim* octeți.

Exemple de date declarate:

Datele se declară în felul următor:
nume: tip valoare_implicită

Exemple:

```
x:.word 5
y:.byte 'a'
str:.asciiz "mesaj"
sum:.space 4
res_byte:.space 1
```

Instrucțiuni pentru transferarea datelor din memorie în registri (și invers):

Instrucțiuni pentru încarcarea valorilor în registri:

lw \$reg, mem_word (load word, încarcă în registrul *reg*, valoarea aflată în memorie în *mem_word*)

lb \$reg, mem_byte (la fel ca mai sus, doar că aici este load byte, deci încarcă un singur byte, nu 4)

la \$reg, adr_mem (load address, încarcă în registrul *reg* adresa de memorie a lui *adr_mem*. Deci instrucțiunile de mai sus încărăcă în registri valoarea de la adresa de memorie, iar această instrucțiune încarcă adresa de memorie. Instrucțiunea aceasta este utilă dacă vrem să iterăm printr-un array. Încarcăm adresa de memorie a array-ului într-un registru, iar apoi incrementăm valoarea din registru. Pentru a accesa valorile din array, folosim *lw* și *lb*, iar ca parametru 2 punem adresa de memorie incrementată)

li \$reg, const (încarcă o valoarea constantă *const* în registrul *reg*)

Instrucțiuni pentru salvarea valorilor din registri în memorie:

sw \$reg, mem (store word, salvează valoarea din registrul *reg* la adresa *mem*)

sb \$reg, mem (store byte, la fel ca mai sus, doar că aici salvează un byte, nu un word)

Exemple de folosire a acestor instrucțiuni:

lw \$t0, x # încarcă în registrul *t0* valoarea din *x* (adică 5)
lb \$t1, y # încarcă în registrul *t1* valoarea din *y* (adică 'a')
la \$t2, str # încarcă în registrul *t2* adresa de memorie a lui *str*
li \$t3, 15 # încarcă în registrul *t3* valoarea constantă 15
sw \$t4, sum # salvează valoarea din registrul *t4* la adresa de memorie a lui *sum*
sb \$t5, res_byte # salvează valoarea din registrul *t5* la adresa de memorie a lui *res_byte* (în cazul asta e un singur byte, nu 4 cum erau la sw)

2.3 Instrucțiuni aritmetice

add/ addu \$dest, \$src1, \$src2 (aduna numerele din registrii *src1* și *src2* și salvează rezultatul în registrul *dest*. Diferența dintre cele două este că add ne poate ajuta să detectăm dacă a fost overflow la adunare printr-un trap, pe când addu este folosit pentru numere unsigned, deci putem verifica dacă rezultatul este mai mare decât cele două numere, iar dacă nu este, e clar că a fost overflow. Nu ne vom concentra pe aceste detalii la tutoriat și probabil vom folosi add în mai toate situațiile)

addi \$dest, \$src, const (adună numărul din registrul *src* cu constanta *const* și salvează rezultatul în registrul *\$dest*)

Observație: Putem face add \$t0, \$t0, \$t1 direct pentru a face operația $t0 = t0 + t1$. Un procesor nu poate face această operație direct, deci în spate, compilatorul de MIPS face niște "magie neagră".

sub/ subu \$dest, \$src1, \$src2 (așemanător cu add/ addu. sub realizează scăderea dintre valorile din registrul *src1* și *src2*, iar rezultatul îl salvează în registrul *dest*. subu este pentru unsigned)

mul \$dest, \$src1, \$src2 (realizează operația $dest = src1 \cdot src2$)

div \$dest, \$src1, \$src2 (realizează operația $dest = src1 \text{ div } src2$. Adică împarte cu rest pe *src1* la *src2* și salvează câtul în registrul *dest*)

rem \$dest, \$src1, \$src2 (realizează operația $dest = src1 \text{ mod } src2$. Adică împarte cu rest pe *src1* la *src2* și salvează restul în registrul *dest*)

2.4 IO + apeluri sistem

Apelurile de sistem nu le putem imagina ca pe niște funcții predefinite de sistemul de operare pe care programul nostru le poate apela pentru a interacționa cu mediul în care se execută. Procesoarele MIPS nefind

folosite pe calculatoare, de regula apelurile de sistem sunt transmise direct catre hardware fără ca vreun sistem de operare să fie intermediar. De exemplu pentru un test de sarcină se poate face un apel de sistem pentru a face un *beep* în cazul în care sunt 2 bare la rezultat. Practic atunci cand facem un apel de sistem pierdem controlul asupra programului, hardware-ul/ sistemul de operare decide când să execute acel apel de sistem, iar apoi decide când să redea controlul programului pentru a-și putea continua executarea (se spune executare, nu execuție, execuție e atunci cand omori pe cineva). Ideea asta de apeluri de sistem este foarte bine să o intelegeți de acum pentru că aşa este în fiecare limbaj de asamblare, nu doar în MIPS, iar în anul 2 va trebui să vă definiți voi propriile apeluri de sistem la cursul de "Sisteme de operare". Dacă vreți să învătați de acum mai multe despre apeluri de sistem, vă puteți uita pe laboratoarele de la cursul de "Sisteme de operare": <https://cs.unibuc.ro/pirofti/so.html>

Pentru a face un apel de sistem este nevoie să încarcăm în registrul \$v0 codul funcției de sistem (nu este ca în C să apelam o funcție după nume, aici fiecare funcție are un număr ca id, care trebuie încărcat în acel registrul.). De regulă se folosește registrul \$a0 pentru a transmite parametrii. După ce valorile necesare au fost setate, se scrie instrucțiunea *syscall*.

Coduri pentru apeluri de sistem

1 - PRINT INT (afisează un număr întreg, se încarcă în \$a0 valoarea de afișat, în \$v0 valoarea 1 (codul pentru apelul se sistem), iar apoi se scrie *syscall*)

4 - PRINT STRING (afisează un sir de caractere, se încarcă în \$a0 adresa de memorie a sirului de caractere, în \$v0 valoarea 4 (codul pentru apelul se sistem), iar apoi se scrie *syscall*)

5 - READ INT (citește un număr de la tastatură, se încarcă în \$v0 valoarea 5 (codul pentru apelul de sistem), apoi se scrie *syscall*. După ce a reușit să citească numărul de la tastatură, sistemul ne va returna valoarea citită în registrul \$v0)

10 - EXIT (dacă facem *li v0, 10*, iar apoi *syscall* este echivalentul lui *return 0*; din C. Încheiem executarea programului.)

In tutorialele viitoare vom aborda și alte apeluri de sistem, dar pentru acest tutoriat, aceste 4 apeluri ne sunt suficiente.

Exemple de apeluri de sistem

```
# PRINT STRING
la $a0, str # încarcă în $a0 adresa sirului de caractere afișat
li $v0, 4
syscall

# READ INT
li $v0, 5
syscall
move $t0, $v0 # în $v0 am primit valoarea citită de la tastatură, deci o copiem în registrul $t0 pentru a lucra cu ea. Sintaxa pentru instrucțiunea move este: move $regd, $regs. Instrucțiunea copiază valoarea din registrul regs în registrul regd.
```

```
# EXIT # echivalentul lui return 0; din C
li $v0, 10
syscall
```

2.5 Structura unui program în MIPS

```
1  # Comentariile în MIPS încep cu #
2  # Ele sunt ignorate de către compilator
3  # Ele sunt valabile până la sfârșitul rândului
4  .data
5  |    # zonă pentru declararea datelor
6  .text
7  |    #cod pentru proceduri
8  main:
9  |    #cod main
10 |    li $v0, 10
11 |    syscall
```

2.6 Exerciții și probleme

Problema 1: Se dă un număr întreg n stocat în memorie, să se calculeze $(n \text{ div } 3) - 2$ și să se afiseze pe ecran rezultatul.

```
1  .data
2  |    n:.word 100756      # numărul declarat în memorie
3  .text
4  main:
5
6  |    lw $t0, n            # incarcă în registrul $t0 valoarea din n
7  |    div $t0, $t0, 3       # realizează împărțirea la 3
8  |    |                    # observați că nu există instrucțiunea divi
9  |    |                    # deci pentru a realiza împărțirea la o constantă
10 |    |                    # este suficient ca al doilea parametru al instrucțiunii
11 |    |                    # să fie o constantă
12 |    |    sub $t0, $t0, 2     # scădem 2
13 |    |    |                  # din nou, nu există subi, deci putem folosi o constantă
14 |    |    |                  # pentru al doilea parametru
15 |    |    move $a0, $t0        # copiază în $a0 valoarea pe care vrem să o afișăm
16 |    |    li $v0, 1             # incarcă în $v0 codul pentru print int
17 |    |    syscall              # realizează apelul de sistem
18 |
19 |    |    li $v0, 10            # exit
20 |    |    syscall
```

2.7 Branch instructions

Sunt instrucțiuni care în urma unei condiții decid dacă să continue executarea la linia curentă sau să continue executarea la altă linie din program. Atunci când calculatorul execută un program și se află la o linie de cod, el ține într-un registru un pointer către linia curentă. Acel pointer poate fi modificat, iar programul să se execute la altă linie. Structura unei instrucțiuni de branch este următoarea: $b_{-} \$src1, \$src2, et.$ b_{-} este instrucțiunea (vom vedea mai jos care sunt instrucțiunile și ce condiții implică fiecare). $\$src1$ și $\$src2$ sunt cele două valori pe baza cărora instrucțiunea decide dacă va continua executarea de la linia de cod la care se află eticheta et .

Instrucțiunile:

bne - (branch not equal, echivalentul lui $!=$ din C. Practic, dacă facem $bne \$src1, \$src2, et$, echivalentul în C ar fi $if(src1 != src2) goto et;$, unde et este un label pus la o linie de cod din program)

beq - (branch equal, echivalentul lui $==$ din C)

ble - (branch less than or equal, echivalentul lui $<=$ din C)

blt - (branch less than, echivalentul lui $<$ din C)

bge - (branch less than or equal, echivalentul lui $>=$ din C)

bgt - (branch greater than, echivalentul lui $>$ din C)

Salt necondiționat: jump (j)

j et (sare la eticheta et direct, necondiționat)

2.8 Exerciții și probleme

Problema 1: Să se afișeze pe ecran toate valorile de la 0 la n-1 (inclusiv) cu n citit de la tastatură.

```

1  .data
2
3      n:.space 4          # numarul nostru (alocam 4 bytes pentru el)
4
5      sp:.asciiz " "      # putem folosi si word in loc
6
7      # (cum am facut la problema anterioara)
8      # spatiu pentru a putea delimita numerele cand
9      # le afisam pe ecran
10
11     .text
12
13     main:
14
15         li $v0, 5          # READ INT
16         syscall
17         move $t0, $v0,      # copiaza in $t0 valoarea citita
18         sw $t0, n           # salveaza in memorie valoarea citita
19
20         li $t1, 0            # $t1 va fi counterul nostrul, deci il initializam cu 0
21         loop:
22             beq $t1, $t0, exit # cand conterul ($t1) este egal cu n (valoarea din $t0),
23             # e clar ca am afisat toate numerele de la 0 la n-1
24             # deci ne ducem la label-ul exit pentru a inchide programul
25             move $a0, $t1        # PRINT INT (copiaza in $a0 valoarea curenta pentru a o afisa)
26             li $v0, 1
27             syscall
28
29             li $v0, 4            # PRINT STRING
30             la $a0, sp          # Incarc in $a0 pointer catre sirul de caractere
31             # care contine un singur spatiu
32             # care contine un spatiu pe ecran pentru a putea delimita
33             # numerele
34             addi $t1, $t1, 1     # incrementeaza conterul
35             j loop              # echivalentul lui i++ din C
36             # continu iterarea
37             # adica continua executarea de la linia
38             # unde este label-ul loop
39         exit:
40             # daca am ajuns aici inseamna ca beq din loop
41             # ne-a adus aici, deci am afisat toate numerele de la
42             # 0 la n-1
43             li $v0, 10           # exit
44             syscall

```

Problema 2: Se citește n de la tastatură. Să se calculeze utilizând structura repetitivă următoarea sumă:

$$\sum_{i=1}^{n-1} (i \text{ div } 3) - 2$$

```

1  .data
2  .text
3  main:
4
5      li $v0, 5
6      syscall
7      move $t0, $v0
8
9      li $t1, 1
10     li $t3, 0
11
12    loop:
13        bge $t1, $t0, exit # am folosit bge pentru a nu avea loop infinit
14        | | | | # în cazul în care primim o valoare mai mică decat 1
15        | | | | # de la intrare
16
17        move $t2, $t1      # calculează termenul  $a_i$  în t2
18        div $t2, $t2, 3
19        sub $t2, $t2, 2
20
21        add $t3, $t3, $t2 # îl adună pe  $a_i$  la sumă
22        | | | | # suma va fi ținută în registrul $t3
23
24        addi $t1, $t1, 1
25
26        j loop
27
28    exit:
29
30    li $v0, 1             # afișează rezultatul sumei
31    move $a0, $t3
32    syscall
33
34    li $v0, 10
35    syscall

```

Problema 3: Se citește $n \in N$ și n numere naturale. Să se calculeze următoarea sumă:

$$\sum_{i=1}^n (a_i \text{ div } 3) - 2$$

, unde a_i este al i-ulea număr citit de la tastatură (după ce s-a citit numărul n).

```

1  .data
2  .text
3  main:
4
5      li $v0, 5
6      syscall
7      move $t0, $v0
8
9      li $t1, 0          # de data asta pornim de la 0,
10     | | | | | | | | | | # nu de la 1 cum era la problema anterioara
11     li $t3, 0
12
13     loop:
14         bge $t1, $t0, exit
15
16         li $v0, 5          # citim termenul a_i și il salvăm în $t2
17         syscall
18         move $t2, $v0
19
20         div $t2, $t2, 3    # nu mai avem nevoie de valoarea citită
21         | | | | | | | | | | # deci putem calcula termenul general direct în $t2
22         sub $t2, $t2, 2
23
24         add $t3, $t3, $t2  # adună termenul calculat la sumă
25
26         addi $t1, $t1, 1
27
28         j loop
29
30     exit:
31
32     li $v0, 1
33     move $a0, $t3
34     syscall
35
36     li $v0, 10
37     syscall

```

Aceasta a fost prima parte din problema care s-a dat anul trecut în prima zi de Advent Of Code. În tutorialele următoare vom mai rezolva probleme date la Advent Of Code în MIPS.

Pont: Faceți probleme de la Advent Of Code în MIPS și în Python pentru a lua 10 la examenele de laborator de la aceste materii. Link problemă: <https://adventofcode.com/2019/day/1>



PROBLEME
CLASICE DE
MIPS CU SCOP
DIDACTIC



ADVENT OF
CODE

References

- [1] Dumitru Daniel Drăgulici. *Curs Arhitectura Sistemelor de Calcul*.
- [2] Larisa Dumitrache. *Tutoriat 2019*
- [3] Bogdan Macovei. *Laboratoare 2019/2020*
- [4] Paul Irofti. *Laborator Sisteme de Operare*
- [5] Advent Of Code. *Day 1 2019*