# 2 Prezentare generala

Limbajele de asamblare reprezinta o imagine a codului masina, iar aparitia lor a fost, in cea mai mare parte, datorata faptului ca programatorii voiau sa lucreze cu un limbaj care poate fi inteles si de oameni, nu doar de un calculator. Prin trecerea de la o insiruire de biti la instructiuni cu sens, s-au putut dezvolta, ulterior, alte limbaje de programare, cunoscute ca limbaje *medium level*, precum *C*, *C++* etc.

O diferenta semnificativa intre limbajele cod masina si asamblare, respectiv limbajele *medium level* si de nivel mai inalt este aceea ca, in ceea ce priveste primele doua, sunt strict dependente de arhitectura in care sunt utilizate. Un limbaj de asamblare depinde strict de procesor, astfel ca *x86* nu poate fi folosit pentru a programa un procesor *MIPS*, ori *RISC-V* etc. Aceasta dependenta stricta nu apare din pricina limbajului de asamblare, ci din cauza codului masina pe care il reflecta. O insiruire de biti are o semnificatie pe un procesor, si poate sa nu aiba sens, ori sa aiba o semnificatie complet diferita pe un alt procesor.

## 2.1 O privire asupra codului masina

In acest caz, ne punem intrebarea - cum stim ce semnificatie are un sir binar pentru procesorul nostru? Cum putem decodifica acea insiruire aparent aleatoare de biti, incat sa intelegem instructiunile de executat? Un raspuns simplu il putem avea daca analizam, in special, clasa procesoarelor *RISC*. Una dintre diferentele considerabile intre *CISC* si *RISC*, este aceea ca in clasa *CISC* instructiunile au o lungime variabila, pe cand in *RISC* toate instructiunile au exact aceeasi lungime. (lungime de reprezentare in biti) De exemplu, daca luam un procesor *RISC* pe 32 de biti, in cadrul acestuia toate operatiile au o codificare binara - cod masina, pe 32 de biti. In clasa *CISC* lucrurile stau complet diferit - operatiile au lungime variabila, si atunci este relativ mai dificil sa parsam.

Vom oferi un exemplu. Sa consideram operatia *add*. In clasa *RISC*, implementarea *add* este mai simpla, intrucat are trei operanzi - *destinatie*, *sursa 1* si *sursa 2*. Putem scrie, astfel `add %7, %8, %11`, unde 7, 8, 11 sunt registrii acestui procesor (registrii sunt identificati sau printr-un nume simbolic, sau printr-un numar asociat).

Reluam operatia oferita: `add %7, %8, %11`. Ne dorim sa ii facem reprezentarea in codul masina. Citind documentatia, obtinem ca formatul instructiunii este urmatorul: (modul in care sunt impartiti cei 32 de biti)

- 6 biti pentru codul operatiei;
- 5 biti pentru codificarea primului registru sursa;
- 5 biti pentru codificarea celui de-al doilea registru sursa; 5 biti pentru codificarea registrului destinatie;
- 5 biti pentru a codifica daca in cadrul operatiei are loc o *shiftare*;
- 6 biti pentru a codifica functia aplicata;

In acest moment, tot ce avem de facut este sa identificam campurile si sa tinem cont de cateva aspecte:

- toate operatiile de format R au codul operatiei 0 - asta inseamna ca avem reprezentarea valorii 0 pe 6 biti;
- codul functiei aplicate - in cazul nostru *add* - este *100000*.

Pentru codificarea registrilor, folosim codificarea in baza 2 pe 5 biti, astfel ca:

- primul registru sursa este `%8`: 8 in baza 2 este 100, iar pe 5 biti este 00100 (adaugam 2 biti nesemnificativi la stanga);
- al doilea registru sursa este `%11`: 11 in baza 2 este 1011, iar pe 5 biti este 01011;
- registrul destinatie este `%7`: 7 in baza 2 este 111, iar pe 5 biti este 00111.

Cum nu avem o operatie de *shiftare*, campurile se vor completa astfel:

- opcode: 000000;
- registru sursa 1: 00100;
- registru sursa 2: 01011;
- registru destinatie: 00111;
- shiftare: 00000;
- functie aplicata: 100000.

Sirul de biti corespunzator este obtinut prin concatenare. Vom concatena si vom grupa si cate 4: `0000 0000 1000 1011 0011 1000 0010 0000`

Acest cod binar este, de fapt, reprezentarea lui `add %7, %8, %11` in cod masina pe un procesor din clasa *RISC*. Putem transforma si in hexa, si vom avea: `00 8B 38 20`. (in general ceea ce vedem cand analizam fisiere binare)

## 2.2 Cod masina pentru un procesor aritmetic

Am vazut care este corespondenta intre codul masina si limbajul de asamblare, si intelegem ca aceasta transformare este bidirectionala - daca avem reguli clare de transformare, atunci avem posibilitatea si de codificare binara, si de decodificare intr-un limbaj de asamblare.

In aceasta tema ne vom imagina ca am reusit sa proiectam un procesor specializat in rezolvarea de **expresii aritmetice**. Vrem ca acest procesor sa fie mult mai expresiv decat cele din clasele *RISC* si *CISC*, incat sa nu ne impuna sa scriem instructiunile pas cu pas, ci vrem sa ii dam operatii complexe pe care sa fie in stare sa le traduca.

Pentru ca avem un procesor, inseamna ca el este programabil printr-un cod masina. Acest cod masina este cel care ar trebui sa transmita Unitatii Aritmetice si Logice a procesorului ce operatii, eventual foarte complexe, ne dorim sa efectueze. Asa cum am vazut si in exemplul de mai sus, avem nevoie de un format clar pentru cum arata instructiunile noastre.

## 2.3 Forma poloneza postfixata

In primul rand, trebuie sa stabilim **ce** vrem sa calculam. Expresiile complexe ne duc cu gandul la scrieri de forma:

$$Exp = ((2 + 125 * 3 + (144 - 11)/27 + 9)/(6 + 3))^2$$

Exista o reprezentare simpla a acestor instructiuni, care sa nu necesite parantezarea. Forma pe care o vom utiliza este **forma poloneza postfixata**. Este postfixata, deoarece operatia este ultima care apare. De exemplu, pentru a calcula $2 + 3$, in forma poloneza postfixata vom scrie 2 3+.

Scrierea este evident de ajutor in operatii complexe. De exemplu, putem scrie 2 3 + 4 * 5 *. Corespondentul este $(2 + 3) * 4 * 5$. Modul in care se evalueaza poate fi implementat cu ajutorul unei stive. De exemplu, pentru 2 3 + 4 * 5 *, adaugam pe stiva 2 si 3, gasim simbolul de adunare, efectuam adunarea, scoatem 2 si 3 de pe stiva si pastram doar rezultatul, respectiv pe 5. Introducem 4 pe stiva, gasim simbolul de inmultire, efectuam inmultirea, scoatem 5 si 4 de pe stiva si pastram 20. Adaugam 5 pe stiva, gasim simbolul de inmultire, efectuam inmultirea, eliminam 20 si 5 de pe stiva, pastram 100. Nu mai avem nimic de executat, deci rezultatul este 100 - elementul din varful stivei.

De exemplu, pentru 15 2 * 4 + mecanismul este similar: adaugam 15 si 2 pe stiva, efectuam inmultirea, eliminam 15 si 2, plasam 30 pe stiva, punem 4 pe stiva, gasim simbolul de adunare, calculam rezultatul, eliminam 30 si 4 de pe stiva, punem 34 si am oprit executarea. Folosim doar operatiile **push** si **pop**, folosim faptul ca operatiile aritmetice accepta doua argumente, si aplicam algoritmul pana cand nu mai avem nimic de parcurs in sir.

Pentru scadere si impartire, daca avem $x\ y\ -$, vom considera $x-y$, iar $xy/$ va fi $x\ /\ y$. Impartirea va fi doar impartirea cu cat si rest, pe intregi, si se va pastra **doar catul**.

Vom considera si operatia *let*, care atribuie o valoare. De exemplu $x$ 1 *let* inseamna ca $x := 1$, iar peste tot unde va fi intalnit $x$ in cadrul expresiei, se va inlocui valoarea lui cu 1.

## 2.4 Formatul instructiunilor

Vom construi codul masina pentru acest procesor. Pentru aceasta, este important de precizat ca vom lucra in stilul unui procesor pe 8 biti (putem lucra cu numere pana la cel mult 255). Formatul instructiunilor noastre va fi putin diferit, pentru ca trebuie sa putem inlantui expresii de complexitatea celor amintite anterior, iar pentru, de exemplu, `-7 18 + 13 / -5 *` nu putem avea instructiuni standard, dar le putem face cu lungime fixa.

Pentru codificarea operanzilor avem urmatoarea structura:

| b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 | b8 | b9 | b10 | b11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | identificator | | bit semn | codificare operand | | | | | | | |

- Codificarea este pe 12 biti (1.5 Bytes);
- semnificatia bitilor este urmatoarea:

– primul bit, $b_0$ este intotdeauna egal cu 1;

– urmatorii 2 biti identifica ce tip are operandul: daca avem 00, atunci operandul este numar, iar daca avem 01, atunci operandul este variabila;

– daca am avut identificatorul de numar, 00, atunci $b_3$ este bit-ul de semn: 0 inseamna ca numarul este considerat pozitiv, respectiv 1 inseamna ca numarul este considerat negativ. Daca am avut identificatorul 01, atunci bit-ul $b_3$ de semn va fi considerat 0;

– operandul poate fi, dupa caz, asa cum am vazut: sau numar pozitiv (de la 0 la 255), sau numar negativ (tot de la 0 la 255, dar cu semn schimbat), sau o variabila - variabilele sunt formate **doar dintr-o singura litera**. De exemplu, daca avem in codificarea operandului `01111000`, 1111000 este, de fapt, 120 in baza 10, care corespunde codului ASCII pentru `x`. Astfel, o codificare completa pe 12 biti `1 01 0 01111000` inseamna ca operandul curent este variabila $x$. Daca am fi avut aceeasi reprezentare in operand, dar cu alt cod de identificator, de exemplu 00, `1 00 0 01111000`, ar fi fost numarul intreg 120, iar daca aveam bitul de semn 1, `1 00 1 01111000`, ar fi fost numarul intreg -120.

Ramane doar sa codificam operatiile. Ele vor respecta o structura similara - un bit initial 1, un identificator, si un cod de operatie aplicata pana la lungimea de 12 biti:

| b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 | b8 | b9 | b10 | b11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | identificator | | codificarea operatiei | | | | | | | | |

Avem urmatoarele codificari:

| Operatie | Codificare |
|---|---|
| let | 000000000 |
| add | 000000001 |
| sub | 000000010 |
| mul | 000000011 |
| div | 000000100 |

Identificatorul este, in acest caz, **10** (cei doi biti care precizeaza identificatorul). Astfel, reprezentarea unei operatii *add* ar fi `1 10 000000001`.

Pentru claritate, vom prezenta identificatorii in urmatorul tabel:

| Identificator | Semnificatie |
|---|---|
| 00 | numar intreg |
| 01 | variabila |
| 10 | operatie |

## 2.5 Un exemplu de translatare

Sa consideram ca vrem sa reprezentam instructiunea `x 1 let x -14 div`.

Vom folosi formatele descrise mai sus, si vom reprezenta pe rand fiecare camp.

1. `x`: cum a fost aratat intr-un exemplu de mai sus, `x` se codifica `1 01 0 01111000`. Vom grupa cate 4 cifre, astfel ca reprezentarea lui `x` este `1010 0111 1000`.

2. il codificam pe `1`. Este o codificare de operand numar intreg pozitiv, deci avem identificatorul `00` si bit-ul de semn `0`. Reprezentarea lui va fi `1 00 0 00000001`, iar pe gruparea cate 4 cifre `1000 0000 0001`.

3. `let` este o operatie, se va codifica precum o operatie, avand deci identificatorul `10`, respectiv codul de operatie `000000000`. Reprezentarea lui va fi `1 10 000000000`, iar pe gruparea cate 4 cifre `1100 0000 0000`.

4. `x` se va reprezenta din nou ca `1010 0111 1000`.

5. `-14` se reprezinta ca operand numar intreg negativ, deci cu identificatorul `00`, bit-ul de semn `1`, iar valoarea `14` in baza $2 \rightarrow$ `1110`, dar pe 8 biti, deci cu 4 biti de 0 la stanga (nesemnificativi): `00001110`. `-14` va fi reprezentat, deci `1 00 1 00001110`, iar pe gruparea cate 4 cifre `1001 0000 1110`.

6. In final, `div` este o operatie, deci identificatorul `10`, si codificarea oepratiei conform tabelului `000000100`, astfel ca va fi `1 10 000000100`, iar pe gruparea cate 4 cifre `1100 0000 0100`.

In acest caz, putem concatena toate reprezentarile binare, si sa avem:

```
1010 0111 1000 1000 0000 0001 1100 0000 0000 1010 0111 1000 1001 0000 1110 1100 0000
0100
```

respectiv in hexa:

```
A7 88 01 C0 0A 78 90 EC 04
```

Avem, astfel, ca operatia din *assembly*-ul nostru, `x 1 let x -14 div`, se translateaza in cod masina in reprezentare hexa in `A7 88 01 C0 0A 78 90 EC 04`.

# 3 Formularea temei

In rezolvarea cerintelor, variabilele utilizate sunt **doar literele mici ale alfabetului limbii engleze**.

## 3.1 Cerinta 1

Cerinta 1 va fi evaluata pe 4 teste si va ajuta sa obtineti un punctaj de 4p din nota 10.

Fie dat ca input un sir hexa, se cere sa se afiseze la *standard output* instructiunea *assembly* de executat.

De exemplu, pentru inputul `A78801C00A7890EC04`, se va afisa la standard output `x 1 let x -14 div`.

## 3.2 Cerinta 2

Cerinta 2 va fi evaluata pe 4 teste si va ajuta sa obtineti un punctaj de 2p din nota 10.

Fie dat ca input o instructiune in limbajul de asamblare al procesorului aritmetic considerat, se cere sa se afiseze la *standard output* evaluarea instructiunii. Pentru aceasta cerinta, in instructiune **nu exista variabile**, ea fiind formata doar din numere intregi si operatii.

De exemplu, poate fi data instructiunea `2 10 mul 5 div 7 6 sub add`. Rezultatul trebuie sa fie conform urmatorului algoritm:

- se adauga 2 pe stiva;
- se adauga 10 pe stiva;
- se identifica operatia mul, se aplica inmultirea dintre 2 si 10, se obtine 20, se elimina 2 si 10 de pe stiva si se pastreaza doar 20;
- se adauga 5 pe stiva;
- se identifica div - actioneaza ca `20 div 5`, iar rezultatul este 4; se elimina 20 si 5 de pe stiva, si se pastreaza doar 4;
- se adauga 7 pe stiva;
- se adauga 6 pe stiva;
- se identifica sub - se calculeaza diferenta dintre 7 si 6, se obtine 1, se elimina 7 si 6 de pe stiva, si se adauga pe stiva valoarea 1. **Atentie!** in acest moment, pe stiva avem 4 (la baza) si 1 in varf, intrucat *sub* este operatie binara si a lucrat doar cu argumentele 7 si 6, dar nu si cu 4 care era deja la baza stivei.
- se identifica add - se calculeaza suma dintre 1 si 4, se obtine 5, se elimina 1 si 4 de pe stiva, se adauga 5;
- am terminat de parcurs sirul, iar rezultatul obtinut este, acum, situat in varful stivei. Rezultatul acestui calcul este 5.

O sugestie de implementare a algoritmului gasiti la finalul acestui document. **Important!** Se cere evaluarea doar pe **unsigned**! **Se garanteaza ca toate operatiile vor fi pe unsigned.**

## 3.3 Cerinta 3

Cerinta 3 va fi evaluata pe 3 teste si va ajuta sa obtineti 1.5p din nota 10.

Fie dat ca input o instructiune in limbajul de asamblare al procesorului aritmetic considerat. Se cere sa se afiseze la *standard output* evaluarea instructiunii. Pentru aceasta cerinta, spre deosebire de cerinta 2, **se folosesc variabile** introduse prin **let**.

Un exemplu de input poate fi `x 1 let 2 x add y 3 let x y add mul`.

Evaluarea va fi facuta astfel:

- se adauga x si 1 pe stiva, este gasit let, si se intelege de acum ca x = 1 in toata expresia aritmetica; sunt eliminati x si 1 de pe stiva;
- se adauga 2 si 1 pe stiva (deoarece acel x este = 1 de acum);
- se intalneste add, se calculeaza suma 3, se elimina 2 si 1 de pe stiva si se pastreaza doar 3;
- se adauga y si 3 pe stiva, este gasit let, si se intelege de acum ca y = 3 in toata expresia aritmetica; sunt eliminati y si 3 de pe stiva;
- se adauga 1 si 3 pe stiva (x, respectiv y);
- se efectueaza adunarea, rezultatul va fi 4, se elimina 1 si 3 de pe stiva, se adauga 4;
- este identificat mul, iar pe stiva aveam deja 3 (de la a treia bulinuta din explicatia curenta) si 4, de la bulinuta anterioara, si se calculeaza rezultatul, 12, se elimina apoi 3 si 4 de pe stiva si se adauga 12;
- nu mai sunt elemente in sir, deci rezultatul final este 12.

Exact ca la cerinta a doua, se garanteaza ca toate operatiile vor fi aplicate pe **unsigned**.

## 3.4 Cerinta 4

Cerinta 4 va fi evaluata pe 5 teste si va ajuta sa obtineti 1.5p din nota 10.

Pentru aceasta cerinta, vom introduce operatii simple de lucru cu matrice. O matrice poate fi reprezentata in forma

```
nrLinii nrColoane nrLinii*nrColoane_elemente
```

Operatiile pe care le putem utiliza pe matrice sunt:

- add - adunam toate elementele din matrice cu valoarea operandului;
- sub - scadem din toate elementele din matrice valoarea operandului;
- mul - inmultim toate elementele din matrice cu valoarea operandului;

- div - impartim toate elementele din matrice la valoarea operandului;
- rot90d - rotim matricea la 90 de grade spre dreapta;

Operatiile pe matrice contin doar instructiunea `let` si una dintre operatiile mentionate anterior. **Nu sunt instructiuni complexe, precum cele de la cerintele anterioare!**

```
x 2 3 1 2 3 4 5 6 let x -2 add
```

In acest caz, matricea `x` este o matrice de `2 linii x 3 coloane`, care are urmatoarea forma:

```
1 2 3
4 5 6
```

Se aplica o adunare cu -2 pe toate elementele matricei:

```
-1 0 1
2  3 4
```

Ca output, la *standard output* se va afisa reprezentarea acestei matrice in forma in care e introdusa in operatie: numarul de linii, numarul de coloane, respectiv elementele matricei, de la stanga la dreapta si de sus in jos. In acest caz, outputul va fi `2 3 -1 0 1 2 3 4`.

Daca avem urmatoarea instructiune: `x 2 3 -1 0 1 2 3 4 let rot90d`, atunci se aplica urmatoarea rotire la dreapta cu 90 de grade:

```
2 -1
3  0
4  1
```

In acest caz, outputul va fi

```
3 2 2 -1 3 0 4 1
```

**Important! Toate operatiile din aceasta cerinta sunt aplicate pe signed!**

# 4 Transmiterea temei si evaluarea

Veti trimite pe mail o arhiva care sa contina patru fisiere `.asm` (sau cate cerinte ati rezolvat), denumite `cerinta1.asm`, `cerinta2.asm`, `cerinta3.asm` si `cerinta4.asm`. Arhiva va avea denumirea `grupa_Nume_Prenume` cu extensia corespunzatoare.

## 4.1 Forma inputului

Pentru fiecare cerinta, inputul va fi doar un sir de caractere, dat pe o singura linie, pe care il veti interpreta ca fiind citit de la tastatura.

Outputul va fi dat **pe o singura linie**: decodificarea instructiunii pentru cerinta 1, respectiv rezultatul evaluarilor pentru celelalte cerinte.

**Important!** Este obligatia studentilor sa se asigure de tratarea corecta a inputului si de afisarea corecta a outputului. Testarea se va face **doar automat**! Punctajul 0 obtinut din cauza unor erori de citire / afisare va ramane 0. **Nu vom evalua manual sursele din cauza unor astfel de probleme.**

# 5 Algoritmul orientativ pentru forma poloneza postfixata

Vom prezenta structura unui algoritm **Forma Poloneza Postfixata** in forma care sa va ajute in implementarea cerintei de evaluare a expresiilor. Pseudocodul de mai jos este similar limbajului `C` si indica ce functii ar trebui sa utilizati.

```
Date de intrare
    - instruction     // instructiune corecta pentru procesorul aritmetic
Date de iesire
    - eval            // evaluarea instructiunii
Algoritm
    res := strtok(instruction, " ")
    firstNumber = atoi(res)
    push firstNumber

    et_loop:
        res := strtok(NULL, " ")

        if res == NULL goto exit

        atoiRes := atoi(res)

        if atoiRes == 0
            // e operatie
            if atoiRes[0] == 'a' // add
                pop x
                pop y
                push (x + y)
            if atoiRes[0] == 's' // sub
                pop x
                pop y
                push (y - x)
            if atoiRes[0] == 'm' // mul
                pop x
                pop y
                push (x * y)
            if atoiRes[0] == 'd' // div
                pop x
                pop y
                push (y / x)
        else
            // este numar; doar il punem pe stiva
            push atoiRes
        goto et_loop

    exit:
        pop eval
        write (eval)
```