Arhitectura Sistemelor de Calcul
Laborator - Partea 0x00

Bogdan Macovei
Ruxandra Bălucea
Cristian Rusu

Octombrie 2023

Cuprins

1 Introducere & Administrativ

1.1 Introducere

Scopul acestui laborator este acela de a va introduce limbajul Assembly x86 si de a va obisnui cu sintaxa AT&T pentru procesoarele x86. In acest sens, accentul este pus, in mare parte, pe insusirea notiunilor elementare de programare intr-un limbaj de asamblare.

Laboratorului va aplica in practica multe dintre principiile arhitecturii calculatoarelor descrise la curs si seminar. Ca programatori, exista multe avantaje daca intelegeti limbajul Assembly:

- veti intelege modul exact in care un calculator executa programele;
- veti putea optimiza la nivel de instructiune codul pe care il scrieti, cand aveti nevoie;
- veti putea intelege ce fac compilatoarele cu codul vostru sursa scris intr-un limbaj de nivel inalt (high level programming language);
- veti intelege modul in care sistemul de operare interactioneaza cu un program scris de voi (executia si oprirea programului, lucrul cu dispozitivele Input/Output prin sistemul de operare);
- veti intelege modul in care sistemul de operare interactioneaza cu hardware-ul vostru (veti putea scriere drivere);
- veti putea integra instructiuni inline-assembly in coduri de C/C++ pentru eficienta si control (veti putea programa sisteme de operare in timp real si sisteme de operare pentru microcontrollere);
- veti putea intelege modul in care functioneaza alte programe pentru care nu aveti disponibil codul sursa (dezasamblarea fisierelor binare uzuale, analiza si diagnoza programelor cu potential risc - virusi, malware, etc.).

Este important sa intelegem si care sunt dezavantajele limbajului Assembly:

- programele scrise in Assembly se pot rula doar pe arhitectura pentru care au fost scrise (machine-specific code), de exemplu programe scrise pentru x86 nu vor rula pe ARM-uri;
- e nevoie de un efort mult mai mare pentru a scrie un program in Assembly (in comparatie cu C/C++, sa nu mai vorbim de Java/C#/python);
- depanarea unui program Assembly mare (multe linii de cod) este extrem de dificila.

Trebuie sa tineti cont de aceste aspect in momentul in care alegeti limbajul de programare in care sa lucrati pentru un anumit proiect.

In acest laborator, cu toate ca accentul va fi pus pe scrierea de la zero a fisierelor .asm, sunt momente in care, pentru optimizare si pentru usurinta citirii codului, vom utiliza instructiuni scrise in limbaj de asamblare doar in anumite portiuni din codurile C/C++.

1.2 Administrativ

Pentru parcurgerea materiei avem alocate 7 laboratoare, dintre care ultimul este rezervat pentru testul final de laborator. Acest document reprezinta prima parte a laboratorului si are ca scop acoperirea activitatilor de la primele 3 laboratoare. Va sugeram sa acoperiti munca, cel putin, in ritmul urmator:

- Laborator 1: instalarea unui sistem de operare Linux, folosirea unui sistem pentru controlul versiunilor (Git), folosirea unui editor de text si folosirea liniei de comanda sub Linux;
- Laborator 2: parcurgerea notiunilor de baza pentru limbajul Assembly x86 descrise in acest document, executia primelor voastre programe Assembly x86 (exemplele din text si 0x00 din sectiunea Exercitii propuse);
- Laborator 3: intelegerea detaliilor descrise in acesta partea a laboratorului si parcurgerea problemelor sugerate la sfarsitul acestui document (toate, sau cat mai multe, probleme din sectiunea Exercitii propuse).

Ultimele 3 laboratoare vor continua lucrul cu limbajul Assembly x86 (notiuni mai avansate) si vor fi acoperite intr-un alt document (Laborator - Partea 0x01).

Nota obtinuta la laborator reprezinta 40% din nota finala, si este formata din trei parti:

1. 10% media testelor de la laboratoarele 2, 3, 4, 5 si 6;
2. 10% o tema, cu predare in luna decembrie sau ianuarie;
3. 20% un test in ultimul laborator.

Nota se calculeaza insumand toate cele trei rezultate si se transmite cu doua zecimale exacte. Pentru promovare, este necesar ca la fiecare dintre cele trei punctaje sa realizati minim 50%.

Prezenta nu se contorizeaza, astfel ca nu afecteaza in niciun mod rezultatul final. De asemenea, nici probleme din sectiunea Exercitii propuse nu se contorizeaza si nu influenteaza nota finala, dar este recomandat sa le lucrati. Testul final de laborator, probleme de la curs si de la examenul final vor fi in stilul problemelor descrise la seminar si laborator.

2 Masina virtuala & Git

Inainte de a incepe programarea in limbajul Assembly trebuie sa ne asiguram ca aveti instalat un sistem de operare corespunzator si aveti ustensilele de programare adecvate pentru scopul nostru.

2.1 Masina virtuala

Acest laborator va fi predat sub Linux, astfel ca daca nu aveti o distributie de Linux ca sistem principal de operare, va trebui sa va instalati cel putin pe o masina virtuala. Recomandarea este sa va luati o imagine de Ubuntu si sa o instalati pe VMWare[1] sau pe VirtualBox[2]. Imaginea sistemului de operare o gasiti pe site-ul de la Ubuntu `https://ubuntu.com/download/desktop`.

Un tutorial video excelent pentru instalarea VirtualBox impreuna cu unele sugestii de setare pentru sistem le gasiti online[3]. Daca nu ati mai folosit Linux, o investitie buna pentru viitorul vostru este un alt tutorial *putin* mai detaliat[4].

2.2 Git

La aceast curs, tot codul sursa scris de voi va fi integrat intr-un sistem pentru controlul versiunilor. Aceasta este o practica standard in industria software. Munca din timpul laboratoarelor va fi salvata pe Git. In acest sens, va recomand sa va faceti un cont pe Github, sa va creati un repository cu numele "Laborator ASC" sau similar, iar toate problemele pe care le lucrati sa le aveti centralizate aici. Cand/daca aveti intrebari legate de programele realizate la acest laborator noi le putem verifica direct online pe Git si va putem sugera solutii.

Un tutorial video excelent pentru instalarea Git pe sistemul vostru este disponibil online[5].

Dupa ce sistemul Git este instalat, suntem gata sa scriem si salvam primul program de test. In terminal, vom scrie

```
mkdir Laborator1
cd Laborator1
git init

git config user.name "Nume Prenume"
git config user.email "email"
```

Vom vrea un fisier de baza in limbajul C,

```
nano file.c
```

In editorul deschis vom scrie primul program C

[1] VMWare, `https://www.vmware.com/products/workstation-player.html`
[2] VirtualBox, `https://www.virtualbox.org/`
[3] How to Use VirtualBox (Beginners Guide), `https://www.youtube.com/watch?v=sB_5fqiysi4`
[4] The Complete Linux Course: Beginner to Power User!, `https://www.youtube.com/watch?v=wBp0Rb-ZJak`
[5] Git & GitHub Tutorial for Beginners, `https://www.youtube.com/playlist?list=PL4cUxeGkcC9goXbgTDQ0n_4TBzOO0ocPR`

```c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Salvam: CTRL+X, Y, Enter.

In continuare, adaugam fisierul pe git si facem commit:

```
git add file.c
git commit -m 'primul commit pe acest proiect'
```

Din browser, salvati linkul cu numele git-ului, trebuie sa fie de forma `https://github.com/nume/numerepo.git`.

```
git remote add origin https://github.com/nume/numerepo.git
git push origin
```

In acest moment aveti fisierul salvat pe git. Verificati online ca fisierul sursa este vizibil.

2.3 Mediul de lucru

La acest curs, la cursuri urmatoare dar si in cariera voastra de programatori, va veti lovi de necesitatea de a folosi un sistem Linux si un sistem de control al versiunilor pentru produse software (de obicei Git). In consecinta, va rugam sa alocati timpul necesar pentru a va instala un mediu de lucru corespunzator unui dezvoltator de software. Trebuie sa aveti un anumit confort in instalarea acestor sisteme pentru ca le veti folosi in continuu de acum incolo, daca alegeti o cariera in IT.

In ultima instanta, daca nu reusiti sa instalati Linux si Git, va recomandam un sistem de programare Assembly avansat in browser[6].

[6] `https://www.onlinegdb.com/` sau `https://ideone.com/`

3 Limbajul Assembly x86

In acest capitol vom acoperi cele mai importante concepte de baza ale limbajului Assembly x86.

x86 este o familie de seturi de instructiuni introdusa o data cu microprocesoarele Intel 8086 (de aici porneste "86" din x86) si 8088. Setul de instructiuni[7] a fost adoptat de alti producatori de microprocesoare (notam in special contributia AMD) si astazi este standardul pentru microprocesoarele pe 64 de biti ale calculatoarelor personale. De-a lungul timpului setul de instructiuni a suferit multe modificari, dar mereu tinand cont de compatibilitatea cu arhitecturile x86 de calcul precedente. Notam aici o diferenta important: telefoanele inteligente si tabletele implementeaza o alta arhitectura de calcul *Advanced RISC Machine (ARM)*, in general.

Assembly x86 este un limbaj de programare de nivel scazut (low-level programming language) care permite controlul direct asupra unitatii centrale de calcul (CPU) a unui calculator cu arhitectura de calcul x86. In multe limbaje de programare de nivel inalt compilatorul creeaza prima data cod assembly ca un pas intermediar. Codul sursa assembly este o bucata de text, deci un procesor nu are cum sa interpreteze si sa execute acest text. Assembly este transformat apoi in cod masina (machine code): instructiuni masina binare codate clar (opcode) care pot fi apoi executate de procesor pas cu pas. Vrem sa va atragem atentia ca Assembly x86 are doua sintaxe distincte: Intel (folosit pe sisteme Windows) si AT&T (folosit predominant pe sisteme Unix). Diferentele intre cele doua nu sunt semnificative (este vorba doar de o sintaxa putin diferita) dar le puteti consulta online[8]. In acest laborator folosim doar sintaxa AT&T. Daca intelegeti una dintre sintaxe, cealalta e foarte usor de citit si inteles.

Componentele principale ale limbajului Assembly x86 sunt:

- Registrii. Ca in orice limbaj de programare, avem nevoie de variable cu care sa lucram. Aceste variable stocheaza date (in cazul nostru, valori pe 32 biti). Putem seta si modifica valorile registrilor si putem realiza operatii cu si intre acesti registri (de exemplu: putem aduna valorile din doi registri si il putem pune intr-un al treilea, sau in unul dintre registrii care deja stocheaza un operand). Unii registri stocheaza o locatie in memorie (de exemplu: trebuie sa stim unde se afla urmatoarea instructiune din program care trebuie executata, deci avem un Instruction Pointer - e defapt un "Instruction Register");
- Instructiuni. Procesorul suporta o serie de operatii intre registrii si locatii din memorie. In primul rand avem nevoie de abilitatea de a muta din/in registri date in/din memorie. In registri, putem face operatii matematice, logice, I/O, etc.
- Flaguri. Dupa ce facem o operatie, avem o serie de flag-uri (indicatori) care se activeaza in functie de rezultat (daca rezultatul operatilor este negativ sau zero, sau daca un overflow s-a produs);
- Adresarea memoriei. In procesor avem doar cativa registri. Daca avem nevoie sa lucram cu un volum mare de date (de exemplu, continutul unui fisier) atunci avem nevoie sa ne putem referi la o adresa din memorie (*Random Access Memory - RAM*). Accesul la registri este rapid, dar citirea/scrierea datelor din/in RAM este o operatie mult mai lenta ca timp;

[7] Intel® 64 and IA-32 Architectures Software Developer Manuals, `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`

[8] `http://asm.sourceforge.net/articles/linasm.html#Syntax`

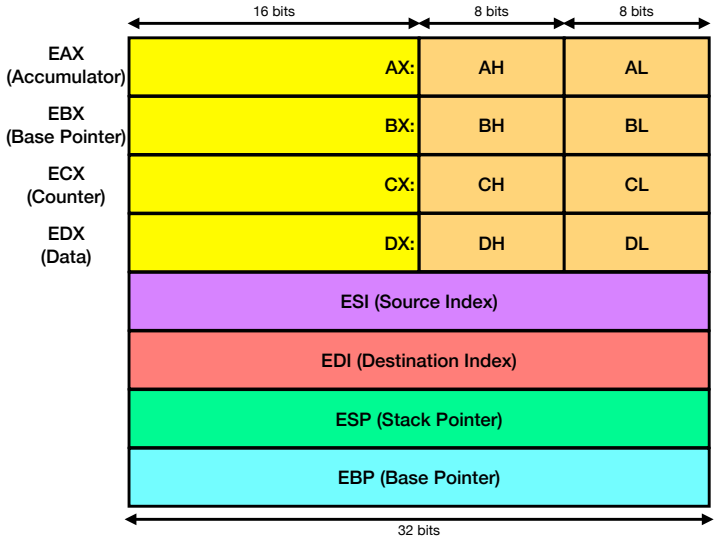
- Stiva programului. Orice program care se afla in executie are o locatie speciala de memorie care se numeste stiva (*the program stack*). Aceasta este o structura de date de tip *Last In First Out (LIFO)*. Vom vedea mai tarziu exemple unde aceasta structura este folositoare (apelul de functii in Assembly);

- Intreruperi. Programul pe care voi il executati (indiferent de limbajul de programare) nu are acces direct la toate resursele hardware. In cele mai multe cazuri, sistemul de operare controleaza accesul la memorie si periferice. De aceea, avem nevoie sa cerem sistemului de operare sa execute niste operatii pentru programul nostru. Aceasta delegare a muncii catre sistemul de operare se realizeaza prin intreruperi.

In continuare, discutam toate aceste componente detaliat.

3.1 Registrii arhitecturii x86

Registrii sistemului x86 pot fi sumarizati in urmatoarea diagrama:



Arhitectura procesorului este pe 32 de biti, astfel ca registrii utilizati sunt tot pe 32 de biti. Un registru reprezinta o cantiate mica de spatiu de stocare pe unitatea centrala de procesare, al carui continut poate fi accesat mai rapid decat datele stocate in alt loc (de exemplu, in memorie sau chiar mai lent pe hard-disk).

Registrii arhitecturii x86 sunt:

- cei 8 din imaginea de mai sus, numiti registrii de uz general (*General-Purpose Registers*);
- un registru pentru *flags*;
- un registru pointer la instructiunea urmatoare ce va fi executata (*Instruction Pointer*);
- alti registri.

3.1.1 Registrii de uz general

Registrul	Nume	Descriere	Restaurat dupa apel
AX	accumulator	este utilizat in operatii aritmetice	nu
BX	base	utilizat ca pointer la date	da
CX	counter	este utilizat in instructiunile repetitive	nu
DX	data	este utilizat in operatii aritmetice si I/O	nu
SP	stack pointer	pointer la varful stivei	da
BP	stack base pointer	pointer la baza stivei	da
SI	source index	pointer la sursa in operatii stream	da
DI	destination index	pointer la destinatie in operatii stream	da

Observatie. Registrul **BP** poate fi utilizat si ca *frame pointer*.

Alta observatie. Urmarim tabelul si observam ca registrii pe arhitectura x86 pe 32 de biti au prefixul **E**. In acest caz, registrul **EAX** este format din **AX** (*accumulator*), **AH** (H - *high*) si **AL** (L - *low*). De exemplu, daca in **EAX** stocam valorea 0, dar punem in **AH** ulterior valoarea 1, atunci intreg registrul **EAX** va stoca valoarea 256 in baza 10. **De ce?**

3.1.2 EFLAGS

Registrii EFLAGS (indicatori) sunt:

- **CF** - *carry flag*, care se seteaza daca dupa ultima operatie a avut loc transport;
- **PF** - *parity flag*, care este setat daca numarul de biti de 1 din rezultatul ultimei operatii este par;
- **ZF** - *zero flag*, care este setat daca rezultatul ultimei operatii a fost 0;
- **SF** - *sign flag*, care este setat daca rezultatul ultimei operatii a fost negativ;
- **OF** - *overlow flag*, care este setat daca rezultatul ultimei operatii a produs overflow.

3.1.3 Adrese

Pentru a reprezenta adresele de memorie, putem utiliza una dintre urmatoarele variante:

- valoarea unui registru pe 32 de biti: (%eax);
- suma dintre constanta numerica si valoarea unui registru: 4(%eax) (inteles ca %eax + 4);
- suma dintre doi registri: (%eax, %edx) (inteles ca %eax + %edx);
- suma dintre doi registri si o constanta numerica: 4(%eax, %edx) (inteles ca %eax + %edx + 4);
- suma a doi registri, dintre care unul inmultit cu 2, 4 sau 8, la care se poate aduna o constanta: 16(%eax, %edx, 4) (inteles ca %eax + 4 * %edx + 16);

3.1.4 Notatie

Este important sa precizam faptul ca, in scrierea programelor, registrii sunt prefixati de simbolul %. Astfel, pentru utilizarea registrului **EAX**, vom scrie *%eax* (vom pastra conventia de notatie cu litere mici).

La fel de important: asa cum registrii sunt precedati de simbolul %, constantele numerice sunt precedate de simbolul $. Vom scrie, astfel, *$4* pentru a reprezenta constanta 4.

3.2 Tipurile de date si structura de baza a unui program

3.2.1 Tipurile de date

Tipurile principale de date pe care le vom utiliza sunt

1. **byte** - dupa cum ii spune si numele, ocupa 1 byte, adica 8 biti in memorie;
2. **single** - ocupa 4 bytes (32 de biti) si este utilizat pentru stocarea numerelor fractionare;
3. **word** - ocupa 2 bytes (16 biti) si este utilizat pentru stocarea intregilor;
4. **long** - ocupa 4 bytes (32 de biti) si este utilizat pentru a stoca intregi pe 32 de biti sau numere fractionare pe 64 de biti. **Double word** (dword) ocupa tot 32 de biti;
5. **quad** - ocupa 8 bytes (64 de biti);
6. **ascii** - este utilizat pentru declararea sirurilor de caractere care nu sunt finalizate cu terminatorul de sir - nerecomandat;
7. **asciz** - este utilizat pentru declararea sirurilor de caractere care sunt finalizate cu terminatorul de sir;
8. **space** - defineste un spatiu in memorie, a carui dimensiune o specificam, de exemplu **.space 4**, insemnand ca se lasa 4 bytes = 32 de biti. Este util atunci cand declaram variabile pe care le calculam in cadrul programului si pentru care nu vrem initial o valoare default.

Pentru a declara o variabila, sintaxa va fi

```
nume: .tip valoare
```

de exemplu

```
x: .word 15
y: .floating 23.74
str: .asciz "String"
```

3.2.2 Structura unui program

Structura de baza a unui program Assembly x86 este:

```
.data
    ;// aici declaram variabilele programului

.text
    ;// de aici incepe codul propriu-zis
    ;// definim main ca fiind o eticheta globala
    ;// reprezentand entry-ul din programul nostru

    ;// in principiu, in acest loc vom scrie procedurile
    ;// aici este scris programul principal
    ;// echivalentul main-ului
.globl main
main:
    ;// instructiuni
```

3.3 Exemplu de program in x86

Important. Instructiunile in limbajul de asamblare au forma

```
[eticheta] mnemonic/operatie [operanzi]
```

3.3.1 Instructiunea MOV

Este o instructiune avand sintaxa

```
mov source, destination
```

cu efectul de a muta sursa in destinatie (atribuire din stanga in dreapta, *source* → *destination*). O informatie interesanta referitoare la aceasta instructiune este faptul ca **mov** e *Turing-complete*[9] (vezi si problema 11 de la Exercitii propuse).

Sursa si destinatia pot fi:

- mov registru, registru; (*mov %eax, %ebx*, i.e., %ebx ← %eax)
- mov adresa de memorie, registru;
- mov registru, adresa de memorie;
- mov constanta numerica, registru (*mov $16, %edx*, i.e., %edx ← 16);

Important. Instructiunea MOV poate fi sufixata de dimensiunea tipului de data, astfel ca putem avea **movl** (pentru long), **movq** (pentru quad) etc. Instructiunile cu sufix permit si o varianta de instructiune in care sursa este o constanta, iar destinatia o adresa de memorie.

3.3.2 Intreruperi de sistem

Pentru a putea scrie un prim program in Assembly, trebuie sa ne asiguram ca ii putem opri executia. Daca in limbajele *medium* si *high-level* de acest lucru se ocupa compilatorul, acum sarcina ii revine programatorului. Prima intrerupere pe care o discutam este **SYSTEM EXIT**.

Pentru a apela o intrerupere, vom folosi instructiunea **int**. Cea mai comuna (cea pe care o vom utiliza cel mai des) este

```
int $0x80
```

care inseamna **system call** (0x80 este hardcodat si este asociat intreruperii care face un apel sistem). System call inseamna ca cerem sistemului de operare sa faca ceva pentru noi (ceva ce programul nostru nu are "drepturi" sa faca direct).

Pentru ca sistemul sa stie de ce este apelat, trebuie sa specificam inca doua lucruri:

- ce functie apelam;
- ce parametri are functia respectiva.

In general, programele C/C++ le finalizam cu un `return 0;`, care este echivalent cu apelul sistem `exit(0)`. Identificam, deci:

[9] Computerphile, `https://www.youtube.com/watch?v=RPQD7-AOjMI`

- functia apelata este **exit**;
- parametrul (argumentul) ei este **0**.

Asa cum si apelul sistemului (system call) este identificat in mod unic prin codul 0x80, si functiile sistem sunt identificate in mod unic prin numere. In acest caz, functiei **exit** ii corespunde numarul 1, iar **registrul** care retine ce functie este apelata este **%eax**. Argumentele sunt date prin urmatorii registri: **%ebx**, **%ecx** etc. (puteti consulta tabelul acesta pentru a vedea parametrii functiilor [10]

Punand aceste informatii cap la cap, intreruperea sistem pentru oprirea programului este:

```
mov $1, %eax
mov $0, %ebx
int $0x80
```

Recapituland:

- cand sistemul depisteaza intreruperea 0x80, intelege ca are un **system call** si isi verifica registrii;
- registrul **%eax** retine codul functiei pe care trebuie sa o apeleze, in acest caz codul **1** inseamna functia **exit**. De unde stim acest cod? Cautam in documentatia sistemului de operare[11];
- stiind ca apeleaza exit care are un argument, se uita in %ebx pentru a citi valoarea acestuia.

Observatie: exista diferite tipuri de apeluri sistem, pe care le vom explica la momentul potrivit. De exemplu, afisarea pe ecran / in fisier text este un apel sistem, citirea de la tastatura / din fisier text este un apel sistem etc.

Observatie: Ce se intampla daca avem un apel de sistem cu multi parametrii? Amintiti-va ca avem un numar mic de registrii. Lucrurile se complica putin daca avem mai mult de 5 parametrii[12].

3.3.3 Un prim program: cod sursa

Avand toate informatiile anterioare, vom scrie primul program complet in limbaj de asamblare:

```
.data
.text
.globl main
main:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Pentru a putea rula programul vom executa urmatoarele comenzi in terminal:

```
gcc --m32 program_exit.asm -o program_exit
```

```
./program_exit
```

Observatie: Pentru a functiona varianta pe 32 de biți ținând cont că sistemul de operare este pe 64 de biți, va trebui să instalați o bibliotecă suplimentară folosind comanda**sudo apt-get install g++-multilib**.

[10] https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit
[11] https://github.com/opennetworklinux/linux-3.8.13/blob/master/arch/sh/include/uapi/asm/unistd_32.h
[12] http://asm.sourceforge.net/articles/linasm.html#Syscalls

3.4 Programe in ASM si Debug

Pentru a putea intelege mai bine cum isi modifica registrii valorile, dar si pentru a putea corecta anumite *bug*-uri cand apar in codul scris, este important sa stim sa utilizam un *debugger*. *Debugging* este o activitate cu care se confrunta orice programator, indiferent de limbajul de programare folosit. Pentru inceput, fie urmatorul program scris in limbaj de asamblare (sa denumim fisierul **program2.asm**):

```
.data
    x: .long 15
.text
.globl main

main:
    movl $0, %eax
    movl %eax, x
    movl x, %ebx

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Executam

```
gcc -m32 program2.asm -o program2
```

Pentru a putea face debug, vom utiliza **gdb**. Rulam in terminal

```
gdb program2
```

Punem un breakpoint la adresa de intrare in program, si anume la **main**:

```
b main
```

Rulam programul

```
run
```

Iar acum, vom executa succesiv

```
i r
stepi
```

(i. e. next step si information registers) pentru a face debug *step by step*. Urmariti cum se modifica valoarea pentru registrii %eax si %ebx.

3.4.1 Un alt program

```
.data
.text
.globl main
main:
```

```
movl $0, %eax
mov $1, %ah

mov $1, %eax
mov $0, %ebx
int $0x80
```

Ce este stocat in registrul %eax dupa executarea instructiunii `mov $1, %ah`? De ce? Care ar fi fost valoarea lui %eax daca am fi avut %al in loc de %ah?

3.5 Hello, world! in Assembly

In aceasta sectiune vom face o intrerupere de sistem pentru afisarea unui sir de caractere pe ecran.

La fel ca pentru **SYSTEM EXIT**, si **PRINT STRING** va avea un cod de functie (in %eax) si un set de argumente (in acest caz sunt 3: %ebx, %ecx si %edx).

Astfel, pentru a putea afisa un string pe ecran, urmarim pasii:

- incarcam in registrul %eax valoarea 4, corespunzatoare functiei *write* (vezi link-ul de mai sus unde sunt listate apelurile de sistem si codurile asociate);
- incarcam in registrul %ebx valoarea 1, corespunzatoare locului in care afisam textul - in consola, adica la **standard output**;
- incarcam in %ecx mesajul de afisat, in acest caz un sir din memorie;
- incarcam in %edx lungimea sirului afisat. Atentie la declararea cu .asciz, pentru ca se considera la lungime si terminatorul de sir.

Fie urmatorul program Assembly complet:

```
.data
    helloWorld: .asciz "Hello, world!\n"
.text
.globl main
main:
    mov $4, %eax
    mov $1, %ebx
    mov $helloWorld, %ecx
    mov $15, %edx
    int $0x80

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

3.6 Operatii aritmetice si logice

3.6.1 Operatii aritmetice

In aceasta parte a laboratorului vom prezenta cele patru operatii aritmetice de baza - adunarea, scaderea, inmultirea si impartirea prin cat si rest.

Instructiune	Efect
add op1, op2	op2 := op2 + op1
sub op1, op2	op2 := op2 - op1
mul op	(edx, eax) := eax × op
imul op	(edx, eax) := eax × op
div op	(edx, eax) := (edx, eax) / op
idiv op	(edx, eax) := (edx, eax) / op

Observatie: pentru `div`, scrierea `(edx, eax)` inseamna ca se imparte $2^{32} * edx + eax$ la `op`. Daca nu vrem sa impartim un numar foarte mare, trebuie sa punem in `edx` valoarea 0 initial.

Explicatii inmultire si impartire.

- atat inmultirea, cat si impartirea au operanzi impliciti, acestia fiind EDX si EAX. Pentru a inmulti un numar cu o anumita valoare (sau a imparti la un numar o anumita valoare), trebuie sa avem deja informatia completata in registrul EAX;
- trebuie sa diferentiem intre MUL si IMUL, intre DIV si IDIV, instructiunea precedata de I fiind pentru prelucrarea numerelor **cu semn**;
- pentru inmultire, rezultatul este dat de $2^{32} \times edx + eax$;
- pentru impartire, eax stocheaza catul, iar edx stocheaza restul;
- in cazul inmultirilor si al impartirilor, operandul *op* **NU** poate fi o constanta numerica! In rest, *op* poate fi registru sau locatie de memorie.

3.6.2 Exemplu

Sa se scrie un program care sa afiseze suma, diferenta, produsul, precum si catul si restul impartirii a doua numere intregi cu semn.

Solutie.

```
.data
    x: .long 30
    y: .long 7
    sum: .space 4
    dif: .space 4
    prod: .space 4
    cat: .space 4
    rest: .space 4
.text
.globl main
```

```
main:
    mov x, %eax
    add y, %eax
    mov %eax, sum

    mov x, %eax
    sub y, %eax
    mov %eax, dif

    mov x, %eax
    mov y, %ebx
    imul %ebx
    mov %eax, prod

    mov x, %eax
    mov y, %ebx
    idiv %ebx
    mov %eax, cat
    mov %edx, rest

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Important. Pentru a inspecta valoarea stocata in variabilele declarate cu **.space 4**, putem folosi tot debugger-ul **gdb**, executand la un pas:

```
print (long) nume_variabila
```

de exemplu,

```
stepi
print (long) sum
```

3.6.3 Operatii logice

Operatiile logice sunt **and**, **or**, **xor** si **not**, primele trei avand structura `operatie sursa, destinatie`, iar **not** avand structura `not destinatie`.

Instructiune	Efect
not op	op := ∼op
and op1, op2	op2 := op2 & op1
or op1, op2	op2 := op2 | op1
xor op1, op2	op2 := op2 ^ op1

Cerinta. Scrieti un program pentru a testa cele patru operatii de mai sus. In loc de **and**, folositi si operatia **test**, si comparati, utilizand debuggerul **gdb**, efectul dintre aceasta si **and**.

3.6.4 Operatii de shift (deplasare) logice

Operatiile logice de deplasare sunt **shr**, **shl**, **sar** si **sal**, toate avand structura `operatie destinatie, numar` unde *numar* este cati biti sunt deplasati. In numele instructiunii, s este *shift*, distinctia l/r este pentru left/right (stanga/dreapta) iar distinctia h/a este pentru logic/arithmetic (deplasare pe biti sau aritmetica). In cazul deplasarilor aritmetice, bitul cel mai semnificativ isi pastreaza valoarea (se pastreaza semnul numarului).

Instructiune	**Efect**
shr numar, op	op := op $\gg$ numar
shl numar, op	op := op $\ll$ numar
sar numar, op	op2 := op $\gg$ numar (cu pastrare semn op)
sal numar, op	op2 := op $\ll$ numar (cu pastrare semn op)

Cerinta. Scrieti un program pentru a testa cele patru operatii de mai sus.

3.7 Salturi conditionate si neconditionate

3.7.1 Salturile neconditionate

In foarte multe cazuri avem nevoie sa putem sari la o anumita instructiune in cod (ganditi-va la if-uri si cicluri repetitive). In aceasta sectiune discutam modurile in care salturile in cod se pot realiza.

Se executa prin instructiunea

```
jmp et
```

sarind la eticheta **et** a programului (se efectueaza un salt in memorie, fara a se tine cont de **IP**).

De exemplu:

```
.data
    text1: .asciz "Text 1\n"
    text2: .asciz "Text 2\n"
.text
.globl main

main:
    jmp et2
    mov $4, %eax
    mov $1, %ebx
    mov $text1, %ecx
    mov $8, %edx
    int $0x80

et2:
    mov $4, %eax
    mov $1, %ebx
    mov $text2, %ecx
    mov $8, %edx
    int $0x80

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

In acest caz, valoarea afisata pe ecran este string-ul “Text 2”.

3.7.2 Salturi conditionate

Instructiunea este

```
j<conditie> et
```

si se executa doar daca este respectata conditia ceruta, sarind la eticheta **et**. In caz contrar, se executa linia urmatoare de cod.

Pot fi testate atat *flag*-urile deja prezentate - carry, overflow, zero, sign, cat pot fi executate si instructiuni corespunzatoare operatorilor relationali ($<$, $<=$, $>$, $>=$, $!=$, $==$).

Operator	Descriere
jc	jump daca este carry setat
jnc	jump daca nu este carry setat
jo	jump daca este overflow setat
jno	jump daca nu este overlow setat
jz	jump daca este zero setat
jnz	jump daca nu este zero setat
js	jump daca este sign setat
jns	jump daca nu este sign setat
Operatori pentru numere fara semn	
jb	jump if below (op1 < op2)
jbe	jump if below or equal (op1 <= op2)
ja	jump if above (op1 > op2)
jae	jump if above or equal (op1 >= op2)
Operatori pentru numere cu semn	
jl	jump if less than (op1 < op2)
jle	jump if less than or equal (op1 <= op2)
jg	jump if greater than (op1 > op2)
jge	jump if greater than or equal (op1 >= op2)
Operatori de egaliate	
je	jump if equal (op1 == op2)
jne	jump if not equal (op1 ! = op2)

Observatie. Pentru a putea utiliza unul dintre operatorii de mai sus, trebuie sa putem compara valori. In acest caz, utilizam instructiunea **cmp**.

Atentie! Daca vom compara **cmp %eax, %ebx**, atunci operatia de $\leq$ trebuie privita ca $\%ebx \leq \%eax$.

```
.data
    a: .long 5
    b: .long -3
    text1: .asciz "a >= b\n"
    text2: .asciz "a < b\n"
.text

.globl main

main:
    mov a, %eax
    mov b, %ebx
    cmp %ebx, %eax
    jge et

    mov $4, %eax
    mov $1, %ebx
    mov $text2, %ecx
    mov $7, %edx
```

```
    int $0x80

    jmp exit

et:
    mov $4, %eax
    mov $1, %ebx
    mov $text1, %ecx
    mov $8, %edx
    int $0x80

exit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Cerinta. Rulati programul de mai sus si executati-l cu debuggerul, pentru a vedea cum se modifica valorile.

3.7.3 Simularea structurilor repetitive

Sa presupunem ca vrem sa simulam urmatorul program dezvoltat in C:

```
#include <stdio.h>

int main()
{
    int n = 5, s = 0;
    for (int i = 0; i < n; i++)
    {
        s += i;
    }
    printf("%d\n", s);
    return 0;
}
```

3.7.4 Simularea structurilor repetitive: solutie 1

In programul Assembly urmator vom vedea cum "simulam" cicluri de calcul cu instructiuni de salt si etichete definite corespunzator.

```
.data
    n: .long 5
    s: .space 4
.text

.globl main

main:
```

```
    mov $0, %ecx
    ;// ecx era registrul-contor

etloop:
    cmp n, %ecx
    je etexit
    ;// daca %ecx == n am terminat parcurgerea

    add %ecx, s
    ;// s += %ecx

    add $1, %ecx
    ;// %ecx += 1

    jmp etloop;
    ;// reiau parcurgerea

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Putem folosi debuggerul pentru a vedea rezultatul. Sa presupunem ca programul se numeste **suma.asm** si am ajuns sa il rulam cu **./suma**. Putem folosi **gdb**-ul astfel:

```
gdb suma
b etexit
run
print (int) s
```

3.7.5 Simularea structurilor repetitive: solutie 2

O solutie mai apropiata de particularitatile de limbaj este cea care utilizeaza *keyword*-ul **loop**. Actiunea acestuia este simpla, se specifica `loop eticheta`, insemnand ca se itereaza in cadrul etichetei date (in programul anterior, aceea era **etloop**), si se opreste executarea cand registrul %ecx devine 0. Atentie la faptul ca %ecx este decrementat automat, nu trebuie sa facem manual aceasta modificare. Rescriem, asadar, programul anterior:

```
.data
    n: .long 5
    s: .space 4
.text

.globl main

main:
    sub $1, n
    mov n, %ecx
```

```
etloop:
    add %ecx, s
    loop etloop
    jmp etexit

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Cerinte:

1. Explicati necesitatea instructiunii `sub $1, n` (verificati documentatia x86 pentru instructiunile *inc* si *dec*);
2. Rulati programul in **gdb** pentru a va asigura de corectitudinea lui.

3.8 Tablouri unidimensionale

Pentru a putea discuta despre tablourile unidimensionale (de intregi, momentan), trebuie sa intelegem adresarea in memorie si sa invatam o noua operatie - Load Effective Address (LEA).

3.8.1 Adresarea in memorie

Fie urmatoarea scriere

```
a(b, c, d)
```

pe care o vom intelege ca fiind $b + c \times d + a$. Mai exact, daca o vom transpune in registri si constante, -8(%ebx, %eax, 4) inseamna $\%ebx + \%eax \times 4 - 8$, iar scrierea

```
mov $5, -8(%ebx, %eax, 4)
```

inseamna ca salvam constanta 5 la adresa de memorie calculata ca fiind $\%ebx + \%eax \times 4 - 8$.

Uneori, anumite informatii pot sa lipseasca, si atunci sa avem, de exemplu, instructiunea

```
mov $0, (%edi, %ecx, 4)
```

Atentie! Cand lucram cu *arrays*, nu putem lasa doar **mov**, ci va trebui sa sufixam cu tipul de date necesar (de exemplu **movl** pentru *arrays* cu elemente de tip **long**).

3.8.2 Load Effective Address (LEA)

Instructiunea **lea** incarca adresa sursei in destinatie, si are forma

```
lea source, destination
```

De exemplu,

```
lea v, %edi
```

incarca in registrul %edi adresa lui v.

3.8.3 Reprezentarea in memorie a tablourilor unidimensionale

Tablourile unidimensionale sunt liniar stocate in memorie, plecand de la o adresa initiala si ocupand, succesiv, atatea locatii de memorie cat ocupa si tipul de date utilizat.

De exemplu, daca tipul de date este **long**, atunci fiecare element al tabloului ocupa o locatie egala cu 4 (long este pe 4 bytes). In acest caz, elementele se vor gasi, relativ la locatia initiala de memorie, astfel:

- primul element, la locatia de memorie initiala, MEM, la care se adauga dimensiunea tipului de date inmultita cu numarul de ordine al elementului, adica MEM + 4 × 0;
- al doilea element se afla la locatia de memorie MEM, la care se adauga produsul dintre 4 (dimensiunea tipului de date) × 1 (indexul curent): MEM + 4 × 1;
- al treilea element se afla la MEM + 4 × 2;
- etc.

Adresa de memorie o vom stoca in registrul %edi (destination index register), iar pentru contorul indexului curent vom utiliza registrul %ecx. In acest caz, observam ca, utilizand scrierea anterioara, in triplet, obtinem elementele sub forma (%edi, %ecx, 4) adica $\%edi + 4 \times \%ecx$.

3.8.4 Declararea tablourilor unidimensionale

Se declara ca o eticheta in memorie, cu elementele insiruite:

```
    v: .long 10, 20, 30, 40, 50
```

si, in general, vom stoca in memorie si numarul de elemente:

```
    n: .long 5
```

3.8.5 Exemple: programul 1

Fie urmatorul program complet Assembly:

```
.data
    n: .long 5
    v: .long 10, 20, 30, 40, 50
.text
.globl main

main:
    lea v, %edi
    mov $0, %ecx
    movl (%edi, %ecx, 4), %edx

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Rulati acest program cu **gdb** si gasiti valoarea stocata in %edx la eticheta **etexit**. Modificati valoarea contorului si observati liniaritatea elementelor in memorie.

3.8.6 Exemple: programul 2

Fie urmatorul program complet Assembly:

```
.data
    n: .long 5
    v: .long 10, 20, 30, 40, 50
.text
.globl main

main:
    lea v, %edi
    mov n, %ecx

etloop:
    mov n, %eax
    sub %ecx, %eax
    movl (%edi, %eax, 4), %edx
```

```
    loop etloop

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Cerinte.

1. Intelegeti ce face acest program, justificand necesitatea registrului %eax.
2. Rulati in **gdb** utilizand instructiunile

   ```
   gdb nume_program
   b etloop
   i r
   c
   ```

   Comanda **c** este utilizata pentru a continua debug-ul, astfel ca va sari mereu in eticheta **etloop**. Cu **i r** putem vedea continutul memoriei, pentru a analiza valorile stocate in %edx.
3. Reusiti sa gasiti valoarea 50 pastrand doar acest breakpoint? Ce ar trebui modificat pentru a putea gasi si %edx = 50?

3.9 Despre inline Assembly

Assembly x86 fi utilizat si pentru a scrie *inline assembly* direct in fisiere C. Tot ce trebuie facut este sa se adauge directiva `__asm__`, in interiorul careia vor fi scrise instructiuni de asamblare, precum in laboratorul curent. In exemplul de mai jos, plecam de la o variabila x si ii adaugam 1 valorii curente pe care o stocheaza.

Observatie. Ignorati, pentru moment, instructiunile *pusha* si *popa*; acestea vor fi explicate in laboratoarele urmatoare. Tot ca observatie, incercati sa mutati variabila x in interiorul functiei *main()* si observati efectul.

Presupunem ca avem urmatorul program, salvat in fisierul **file.c**:

```
#include <stdio.h>
int x = 1;
int main()
{
    __asm__
    (
        "pusha;"

        "mov x, %eax;"
        "add $1, %eax;"
        "mov %eax, x;"

        "popa;"
    );

    printf("%d\n", x);
    return 0;
}
```

Programul se va compila si executa cu

```
gcc file.c -o file -m32
./file
```

Cerinta. Implementati si alte programe din acest suport de laborator, utilizand **inline assembly**.

4 Exercitii propuse

4.1 Laboratorul 1

1. Se dau doua numere naturale **x** si **y**. Sa se scrie un program in asamblare care sa realizeze interschimbarea lor. Sa se observe efectul folosind debugger-ul.

2. Fie **s** un spatiu alocat de 12 octeti. Folosind codul functiei read (vezi link-ul de mai sus unde sunt listate apelurile de sistem si codurile asociate) si stiind ca regula de incarcare este aceeasi ca in cazul functiei write, cititi in **s** de la tastatura (codul pentru stdin este 0) 12 caractere. Afisati-l pe **s**.

4.2 Laboratorul 2

1. Fie **x**, **y** doua numere natruale salvate in memorie. Sa se scrie un program care calculeaza in doua moduri (x/16) + (y $\times$ 16).

2. Sa se adauge in cadrul programului anterior la final un pas pentru a verifica daca cele doua valori calculate sunt egale. Daca da, sa se afiseze "PASS", altfel "FAIL".

3. Fie **a, b, c** trei numere salvate in memorie si **min** un spatiu alocat de 4 octeti. Sa se salveze in **min** cel mai mic numar dintre cele trei.

4. Sa se verifice daca un numar dat este prim (folositi ambele variante de structuri repetitive).

4.3 Laboratorul 3

1. Sa se determine maximul si numarul de aparitii al acestuia intr-un array.

2. Fie **s** un string salvat in memorie si **t** un spatiu alocat cu aceeasi numar de octeti. Sa se obtina in **t** inversul string-ului **s** si sa se afiseze pe ecran.

5 Exercitii suplimentare

1. Scrieti secvente de cod Assembly x86 care calculeaza urmatoarele:
   (a) EAX $\leftarrow$ EAX & 1
   (b) EAX $\leftarrow$ EAX ^ EAX
   (c) EAX $\leftarrow$ (EAX ^ EBX) ^ EBX
   (d) EAX $\leftarrow$ EAX & (EAX $-$ 1)
   (e) EAX $\leftarrow$ (EAX & $\sim$EBX) | ($\sim$EAX & EBX)
   (f) EAX $\leftarrow$ (EAX & $\sim$EAX) | ($\sim$EBX & EBX)
   (g) EAX $\leftarrow$ (EAX | $\sim$EBX) & ($\sim$EAX | EBX)
   (h) EAX $\leftarrow$ (EAX | $\sim$EAX) & ($\sim$EBX | EBX)
   (i) EAX $\leftarrow$ EBX ^ ((EAX ^ EBX) & $-$(EAX $<$ EBX))
   (j) EAX $\leftarrow$ EAX ^ ((EAX ^ EBX) & $-$(EAX $<$ EBX))
   (k) ECX $\leftarrow$ EAX + EBX - ECX
   (l) EBX $\leftarrow$ ((EAX $\times$ 2 + EBX) $\times$ 2 + ECX)/2
   (m) EAX $\leftarrow$ (EAX + EBX + ECX) $\times$ 16
   (n) ECX $\leftarrow$ (EAX/16) + (EBX $\times$ 16)
   (o) ECX $\leftarrow$ (EAX & 0xFF00) + (EBX & 0x00FF)
   (p) EAX $\leftarrow$ (EAX + EBX)/(EAX + ECX)
   (q) EAX $\leftarrow$ (EAX$\times$EBX)/(ECX^2)
   (r) EAX $\leftarrow$ EAX$\times$((EAX+EBX)/ECX+1)
   (s) EAX $\leftarrow$ 1 + EAX + EAX^2/2 + EAX^3/6 + EAX^4/24 + ...
2. In registrul EAX avem un numar intreg. Verificati daca numarul este negativ, pozitiv, sau zero.
3. In registrii EAX si EBX avem doua valori distincte. Implementati SWAP(EAX, EBX) folosind doar functii logice si niciun registru suplimentar sau memorie.
4. In registrul EAX aveti un numar natural. Gasiti cea mai apropiata putere de 2 mai mare decat acest numar. (verificati daca x86 are deja o instructiune care realizeaza aceasta operatie, daca da atunci ganditi-va de ce este importanta aceasta operatie)
5. Se da un numar natural in registrul EAX, numarati cati biti de "1" sunt in reprezentarea binara a numarului. (verificati daca x86 are deja o instructiune care realizeaza aceasta operatie, daca da atunci ganditi-va de ce este importanta aceasta operatie)
6. Se da un numar natural strict pozitiv in registrul EAX, calculati $\lfloor \log_2 \text{EAX} \rfloor$. (verificati daca x86 are deja o instructiune care realizeaza aceasta operatie, daca da atunci ganditi-va de ce este importanta aceasta operatie)
7. Se da un numar natural in registrul EAX, calculati dimensiunea celui mai lung sir de biti "1" consecutivi in reprezentarea binara a numarului.

8. Se da un numar natural in registrul EAX, verificati daca numarul este un palindrom (in reprezentarea binara).

9. Se da un numar intreg cu semn, sa se determine toti divizorii acestuia.

10. Utilizand doar instructiuni **mov**, sa se obtina intr-un registru numarul 2318. (fara a utiliza instructiunea `mov $2318, %eax`)

11. Sa se determine cel de-al n-lea termen al sirului Fibonacci, unde n este dat de la tastatura. Nu se garanteaza ca este un numar intreg pozitiv.

12. Sa se calculeze suma dintr-un tablou unidimensional. Sa se calculeze si media aritmetica (impartire prin cat si rest) a elementelor din tablou.

13. Calculati EAX $\leftarrow$ EAX! (EAX factorial).

14. Sa se determine cele mai mici doua elemente dintr-un array, utilizand o singura parcurgere.

15. Fie dat un array in memorie de lungime impara. Stim ca toate elementele sunt duplicate, cu exceptia unuia, de exemplu `v = { 1245, 342, 543523, 342, 4234, 1245, 543523 }`, unde elementul `4234` este elementul singular. Sa se scrie un algoritm eficient care sa determine acest element (complexitate spatiu $O(1)$ si complexitate timp $O(n)$, unde n este lungimea array-ului).

16. Utilizand algoritmul de cautare binara, sa se verifice daca un array contine un anumit element. Daca da, se va determina pozitia acelui element, altfel pozitia va fi -1.

19*. Sa se calculeze radicalul intreg al unui numar (integer square root)[13]. (pentru un calcul aproximativ rapid al radicalului vedeti o secventa de cod ajunsa celebra: codul sursa pentru Q_rsqrt din Quake[14])

20*. Sa se sorteze un tablou unidimensional. Sa se scrie un program C care foloseste cat mai multe instructiuni scrise inline assembly pentru a sorta un tablou unidimensional (ASM pur si inline ASM).

21*. Pe urmele primului programator, Ada Lovelace, sa se determine cel de-al n-lea termen al sirului Bernoulli[15].

[13] `https://en.wikipedia.org/wiki/Integer_square_root`

[14] Q_rsqrt, `https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c`

[15] `https://writings.stephenwolfram.com/2015/12/untangling-the-tale-of-ada-lovelace/`

6 Resurse suplimentare

Tutoriale si explicatii suplimentare se gasesc la urmatoarele resurse online.

Pentru sintaxa AT&T:

1. Adam Ferrari, x86 Assembly Guide[16];
2. Alexander Mishurov, AT&T assembly syntax and IA-32 instructions[17].

Pentru sintaxa Intel:

1. Vlad Radulescu, Laboratoare ASM[18];
2. David Evans, x86 Assembly Guide[19];
3. Cristian Botau, Introducere in asamblare[20].

[16] http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html
[17] https://gist.github.com/mishurov/6bcf04df329973c15044
[18] https://sites.google.com/view/danton-asm/asm1?authuser=0
[19] http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
[20] https://www.infoarena.ro/introducere-in-asamblare