Bune practici

Mai jos găsiți un ghid bune practici, recomandări și greșeli frecvente care apar în momentul în care lucrați în limbajul de asamblare. Să țineți cont, vă rugăm, de acestea în momentul în care lucrați în laboratoare sau teme de casă.

Exemple

- Program care afișează `"Hello, World!"` folosind asamblare cu NASM în linia de comandă (x86, 32 de biți)
  - https://gist.github.com/razvand/782d6471f23352300f11
- Program care afișează `"Hello, World!"` folosind asamblare cu NASM în linia de comandă (x86, 64 de biți)
  - https://gist.github.com/razvand/49aa3bbc13ee29f4f46b
- Program care apelează funcții externe pe Windows (x86, 32 de biți)
  - https://gist.github.com/razvand/a9dbb356d2344723408c
- Program care apelează funcții externe pe Linux (x86, 32 de biți)
  - https://gist.github.com/razvand/65dc30fbb64b37f4d617

Erori des întâlnite

Confuzii la accesarea datelor în memorie (operatorul de dereferenţiere)

Pentru cei care sunt la început de drum la a învăţa assembly, este o confunzie foarte mare cum se foloseşte operatorul de dereferenţiere din asamblare: `[ ]`
Care este diferenţa între `op reg, var` şi `op reg, [var]`?
În 99.999999999999% din cazuri, operaţia fără paranteze pătrate înseamnă să foloseşti adresa acelei variabile pe post de operand. Exemple:

```
section .data
    var: DD 34
section .text
    mov eax, var ; put var's >>address<< into the eax register
    add eax, var ; add to eax, the >>address<< of var
```

Acest cod este echivalent cu următorul cod din **C**:

```
int var = 34;
eax = &var; /* mov eax, var */
eax = eax + &var; /* add eax, var */
```

În cazul în care foloseşti paranteze pătrate:

```
section .data
    var: DD 34
```

```
section .text
    mov eax, [var] ; put var's >>value<< into eax
    add eax, [var] ; add to eax, the >>value<< of var
```

Acest lucru ar fi echivalent în **C** cu:

```
int var = 34;
eax = var; /* mov eax, [var] */
eax = eax + var; /* add eax, [var] */
```

Printre singurele instrucţiuni care fac abatare de la aceste reguli, este **lea** (load effective address).

```
section .data
    var: DD 34
section .text
    lea eax, [var] ; put var's >>address<< into the eax register
```

În rest, toate celelalte instrucţiuni aderă la regulile enunţate mai sus. Dacă or mai exista şi alte instrucţiuni care se comportă ca **lea**, cel mai probabil nu vor fi tratate în aceste laboratoare.

Încărcarea datelor în registre

Adesea apar erori chiar la încărcarea datelor în registre.

load.asm

```
extern printf

section .data
    nr: DB 23
    str: DB 'number: %d',

section .text

global main

main:
    mov eax, [nr]
    push eax
    push str
    call printf
    add esp, 8
    ret
```

În momentul în care se face `mov eax, [nr]`, instrucţiunea **mov** încearcă să deducă dimensiunea mutării (câte date/bytes să ia de la adresa de la care începe **nr**?). **nr** fiind doar o adresă în memorie, nu-i spune nimica compilatorului. Din acest motiv, compilatorul încearcă să se uite dacă nu cumva în această instrucţiune nu există şi un registru implicat. Îl vede pe **eax**. În consecinţă, compilatorul va

codifica instrucţiunea astfel încât în **eax** să se aducă sizeof(eax) (adică 4 bytes) de la adresa lui **nr**. Deşi **nr** are valoarea 23, programul afişează **number: 1836412439**.
De ce? Pentru că la **nr** find un singur byte, procesorul continuă să aducă din memorie încă 3 bytes astfel încât să îl poate umple pe **eax**. În cazul nostru, după **nr**, în memorie, este declarat vectorul **str**, aşa că va lua încă 3 bytes de la el pentru a-l umple pe **eax**.

Intuitiv, v-aţi aştepta ca asamblorul/compilatorul să urle la voi "că uite domne, eu am declarat variabila de 1 byte, şi am scris din greşeală că vreau să aduc 4 bytes de acolo". Ca şi în cazul limbajului **C**, limbajul de asamblare te lasă să te împuşti singur în picior. Nu este treaba lui să facă check-uri. Dacă tu vrei **1 milion de bytes** de la adresa **0xB00B5**, el o să-ţi codifice programul în binar astfel încât să-ţi aducă date de la adresa **0xB00B5**. Că după îţi bubuie programul în faţă cu un **Segmentation Fault** pentru că ai încercat să accesezi o zonă de memorie care nu ţi-a fost alocată, e deja treaba sistemului de operare şi a procesorului.

O primă rezolvare

O primă încercare de a rezolva problema ar fi să încercăm să-l aducem pe **nr** direct într-un registru de 1 byte.

load_byte.asm

```
extern printf

section .data
    nr: DB 23
    str: DB 'number: %d',

section .text

global main

main:
    mov al, [nr] ; modified line
    push eax
    push str
    call printf
    add esp, 8
    ret
```

Mie, personal, s-a întâmplat ca acum să-mi dea corect afişarea. Dar programul nu este încă corect. Noi îi transmitem lui **printf** să afişeze un număr reprezentat pe 4 bytes. Deşi noi am încărcat datele în **al**, noi îi spunem lui **printf** să afişeze conţinutul la tot **eax**, nu doar la **al**. În unele cazuri, s-ar putea ca conţinutul părţii superioare a lui **eax** să nu fie curat, din cauza codului care s-a executat anterior. S-ar putea ca cei mai semnificativi 3 bytes să fie plini cu garbage (date random), şi afişarea noastră tot să nu fie corectă. Astfel că mai este nevoie de încă o corectură:

load_byte.asm

```
extern printf

section .data
    nr: DB 23
    str: DB 'number: %d',

section .text

global main

main:
    xor eax, eax ; eax = 0
    mov al, [nr] ; modified line
    push eax
    push str
    call printf
    add esp, 8
    ret
```

Tot registrul **eax** trebuie iniţializat la , ca să fim singuri că nu există junk în partea superioară.

Cum să eviţi să te împuşti singur în picior ?

Există un set de cuvinte cheie în **NASM** care îi specifică asamblorului/compilatorului **pe câţi bytes** are loc operaţia. Acestea sunt: byte, word şi dword (double word).

load.asm

```
extern printf

section .data
    nr: DW 23 ; declare a variable of word type (2 bytes)
    str: DB 'number: %d',

section .text

global main

main:
    mov eax, word [nr] ; try to access a varible of word type ; try to
bring 2 bytes into eax
    push eax
    push str
    call printf
    add esp, 8
    ret
```

Dacă de exemplu ai declarat un vector/variabilă de words, peste tot unde se accesează un element din acel vector/varibilă prefixează accesul cu tipul variabilei (byte, word, dword, etc.). În felul acesta, asamblorul îţi va da o eroare sugestivă prin care să-ţi dai seama că codul tău nu este tocmai în regulă:

```
arcade@Arcade-PC:~/workspace/asm_exemple > nasm -f elf32 load.asm
load.asm:12: error: invalid combination of opcode and operands
```

Poate că nu ai un cod care compilează, dar măcar nu ai un cod care compilează şi ruleaza greşit.

Segmentation Fault debugging: GDB quicky

gdb este un debugger în linie de comandă. Unul din lucrurile la care ne poate ajuta acesta este să găsim punctele în care ne dă **Segmentation Fault** un program. Mulţi abordează această problemă prin imbricarea de **printf**-uri în puncte intermediare în program. Acest lucru nu prea ajută. Uitaţi cam cum este prelucrat un program de un procesor:

1. Într-o singură etapă se aduc mai multe instrucţiuni din memorie. Accesul la memorie este scump, şi dacă la fiecare instrucţiune de 5-6 bytes ne-am duce în memorie, nu am avea o performanţă foarte bună. Din acest motiv s-a inventat un modul în procesor, numit prefetching, în care se înmagazinează mai multe instrucţiuni de la adresa de la care se aduce cod/instrucţiuni, pentru ca execuţia să fie mai fluidă.
2. În momentul în care procesorul îşi dă seama că una din instrucţiuni accesează o zonă nevalidă din memorie, trimite un semnal către sistemul de operare. Şi sistemul de operare este tot o bucată de cod care se execută pe procesor. Până când acest semnal trezeşte codul din sistemul de operare, e foarte posibil ca programul să mai fi executat o căruţă de instrucţiuni, din acest motiv, o înşiruire de printf-uri s-ar putea executa şi după instrucţiunea care a produs Segmentation Fault-ul.
3. Sistemul de operare se trezeşte şi închide forţat programul care a cauzat probleme. Printre datele primite de la semnal se regăseşte şi adresa instrucţiunii care a cauzat Segmentation Fault. Cu un debugger, se poate afla şi din userspace ce instrucţiune a cauzat Segmentation Fault.

Exemplu de cod cu probleme:

segfault.asm

```
extern printf

section .data
    str: DB `number: %d\n`
    nr: DD 1, 2, 3, 4, 5
    len: DD 4000

section .text

global main

main:
    xor ecx, ecx
```

```
keep_printing:
    push ecx ; save ecx, because it will be destroyed by printf call
    push dword [nr + 4*ecx]
    push str
    call printf
    add esp, 8
    pop ecx ; restore ecx
    inc ecx
    cmp ecx, [len]
    jl keep_printing
    ret
```

Programul parcurge un vector şi afişează valorile sale. Deşi programul are doar 5 elemente, **len**-ul este setat greşit la 4000 de elemente. Dacă compilăm şi rulam programul acesta ne va da un **segfault**:

```
# ...
number:
number:
number:
Segmentation fault
```

Cum rulăm gdb:

```
gdb nume_binar
```

Exemplu:

```
catalin.vasile3004@fep ~ $ gdb ./segfault
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /export/home/acs/stud/c/catalin.vasile3004/load...(no
debugging symbols found)...done.
(gdb)
```

În acest moment s-a deschis consola debugger-ului, **dar programul NU rulează**. Pentru a rula programul:

```
set disassembly-flavor intel
run param1 param2 param3 < fisier.in > fisier.out
```

Cu **run**-ul dat ca exemplu, e ca şi cum am fi rulat programul în felul următor:

```
./segfault param1 param2 param3 < fisier.in > fisier.out
```

`set disassembly-flavor intel` vă ajută pentru a afişa eventualele printări de cod de asamblare într-o sintaxă cunoscută. Limbajul de asamblare reprezintă un set de alias-uri pentru instrucţiunile din binarul unui program. Aceste alias-uri nu au o formă standardizată motiv pentru care acestea diferă de la un asamblor la altul. By default, tool-urile din Linux folosesc sintaxa AT&T. 99% din tool-urile din Linux (gdb NU se află printre ele) pot primii argumentul `-M intel` pentru a afişa sau a trata codul de asamblare ca şi cum ar fi în sintaxa recomandată de Intel (care se regăseşte şi la NASM). Programe care pot primi acest flag sunt: gcc (gas), objdump, etc.
Revenind la gdb, în momentul în care rulăm o să ne dea următoarea eroare:

```
# ...
number:
number:
number:
number:

Program received signal SIGSEGV, Segmentation fault.
0x08048423 in keep_printing ()
```

Pentru a vedea ce instrucţiunea a provocat segfault, putem da următoarea comandă:

```
(gdb) display/10i $pc
1: x/10i $pc
=> 0x8048423 <keep_printing+1>:    push   DWORD PTR [ecx*4+0x804a02c]
   0x804842a <keep_printing+8>:    push   0x804a020
   0x804842f <keep_printing+13>:   call   0x80482f0 <printf@plt>
   0x8048434 <keep_printing+18>:   add    esp,0x8
   0x8048437 <keep_printing+21>:   pop    ecx
   0x8048438 <keep_printing+22>:   inc    ecx
   0x8048439 <keep_printing+23>:   cmp    ecx,DWORD PTR ds:0x804a040
   0x804843f <keep_printing+29>:   jl     0x8048422 <keep_printing>
   0x8048441 <keep_printing+31>:   ret
   0x8048442 <keep_printing+32>:   xchg   ax,ax
```

- **$pc** este o variabilă **gdb**, şi vine de la **P**rogram **C**ounter (este pointer-ul la instrucţiunea curentă).
- **display** face dump de la un pointer dat ca argument, în cazul nostru **$pc**
- **i**-ul îi spune lui **display** să interpreteze datele de acolo ca şi cum ar fi instrucţiuni
- **10** îi spune lui **display** câţi operanzi de tipul **i** (instrucţiune) să afişeze

  Prin <keep_printing+some_number>, **gdb** incearcă să ne arate cam pe unde ar fi această instrucţiune. În cazul nostru instrucţiunea este aproape de label-ul **keep_printing**.
  Pentru a vedea ce valoare a avut un registru la momentul în care s-a declanşat **segfault**-ul, puteţi da:

  ```
  (gdb) print $nume_registru
  ```

  În cazul nostru s-ar putea să ne intereseze ce valoare are **ecx**. Pentru a afla acest lucru:

```
(gdb) print $ecx
```

Categorie 3

- TODO
- TODO