Laborator 05: Introducere în limbajul de asamblare

În acest laborator, vom prezenta o parte din instrucțiunile x86, precum și o suită de exemple introductive.

Introducere

Înainte de a începe efectiv să învățăm să citim cod scris în limbaj de asamblare, iar apoi să scriem primele noastre programe, trebuie să răspundem la câteva întrebări.

Ce este un limbaj de asamblare?

După cum probabil știți, rolul de bază al unui calculator - în speță, al procesorului - este de a citi, interpreta și executa instrucțiuni. Aceste instrucțiuni sunt codificate în cod mașină.

Un exemplu ar fi:

```
10110000000011000110011000110001110111111111111100100
```

Această secvență de biți nu ne spune nimic în mod deosebit. Putem să facem o conversie în baza 16 pentru a o comprima și grupa mai bine.

```
\xB0\x0C\x66\x31\xD2\xFF\xE4
```

În continuare, pentru mulți dintre noi nu spune nimic această secvență. De aici vine necesitatea unui limbaj mai ușor de înțeles și utilizat.

Limbajul de asamblare ne permite să scriem programe text care mai departe vor fi traduse, prin intermediul unui utilitar numit **asamblor**, specific fiecărei arhitecturi, în cod mașină. Majoritatea limbajelor de asamblare asigură o corespondență directă între instrucțiuni. De exemplu:

```
mov al, 12 <-> '\xB0\x0C'
xor dx, dx <-> '\x66\x31\xD2'
jmp esp    <-> '\xFF\xE4'
```

Deoarece limbajul de asamblare depinde de arhitectură, în general nu este portabil. De aceea, producătorii de procesoare au încercat să păstreze neschimbate instrucțiunile de la o generație la alta, adăugându-le pe cele noi, pentru a păstra măcar compatibilitatea în cadrul aceleiași familii de procesoare (de exemplu, procesoarele Intel 80286, 80386, 80486 etc. fac parte din genericul Intel x86).

De ce să învăț limbaj de asamblare?

Pe lângă valoarea didactică foarte mare, în care înțelegeți în ce constă "stack overflow", reprezentarea datelor și ce e specific procesorului cu care lucrați, există câteva aplicații în care cunoașterea limbajului de asamblare și, implicit, a arhitecturii sunt necesare sau chiar critice.

Debugging

Este destul de probabil ca cel puțin unul din programele pe care le-ați scris în trecut să genereze următorul rezultat:

```
Segmentation fault
```

Uneori, veți fi întâmpinați de o serie de date similare cu cele de mai jos:

```
Page Fault cr2=10000000 at eip e75; flags=6
eax=00000030 ebx=00000000 ecx=0000000c edx=00000000
esi=0001a44a edi=00000000 ebp=00000000 esp=00002672
cs=18 ds=38 es=af fs=0 gs=0 ss=20 error=0002
```

Pentru cineva care cunoaște limbaj de asamblare, e relativ ușor să se apuce să depaneze problema folosind un debugger precum gdb [http://www.gnu.org/software/gdb/] sau OllyDbg [http://www.ollydbg.de/], deoarece mesajul îi furnizează aproape toate informațiile de care are nevoie.

Optimizare de cod

Gândiți-vă cum ați scrie un program C care să realizeze criptare și decriptare AES [http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf]. Apoi, indicați compilatorului faptul că doriți să vă optimizeze codul. Evaluați performanța codului respectiv (dimensiune, timp de execuție, număr de instrucțiuni de salt etc.). Deși compilatoarele sunt deseori trecute la categoria "magie neagră", există situații în care pur și simplu știți ceva [https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf] despre procesorul pe care lucrați mai bine ca acestea.

Mai mult, e suficient să înțelegeți cod asamblare pentru a putea evalua un cod și optimiza secțiunile critice ale acestuia. Chiar dacă nu veți programa în limbaj de asamblare, veți fi conștienți de codul ce va fi generat de pe urma instrucțiunilor C pe care le folosiți.

Reverse engineering

O mare parte din aplicațiile uzuale sunt closed-source. Tot ce aveți când vine vorba de aceste aplicații este un fișier deja compilat, binar. Există posibilitatea ca unele dintre acestea să conțină cod malițios, caz în care trebuie analizate într-un mediu controlat (malware analysis/research).

Embedded și altele

Există cazuri în care se impun constrângeri asupra dimensiunii codului și/sau datelor, cum este cazul device-urilor specializate pentru un singur task, având puțină memorie. Din această categorie fac parte și driverele pentru dispozitive.

Fun

Pentru mai multe detalii, discutați cu asistentul vostru de laborator pentru a vă împărtăși experiența lui personală în materie de limbaj de asamblare și cazurile practice de utilizare folosite.

Familia x86

Aproape toate procesoarele importante de la Intel împart un ISA (instruction set architecture) comun. Aceste procesoare sunt puternic backwards compatible, având mare parte din instrucțiuni neschimbate de-a lungul generațiilor, ci doar adăugate sau extinse.

Un ISA definește instrucțiunile pe care le suportă un procesor, dimensiunea registrelor, moduri de adresare, tipurile de date, formatul instrucțiunilor, întreruperile și organizarea memoriei.

Procesoarele din această familie intră în categoria largă de CISC (Complex Instruction Set Computers). Filozofia din spatele lor este de a avea un număr mare de instrucțiuni, cu lungime variabilă, capabile să efectueze operații complexe, în mai mulți cicli de ceas.

Registre

Unitățile de lucru de bază pentru procesoarele x86 sunt registrele. Acestea sunt o suită de locații în cadrul procesorului prin intermediul cărora acesta interacționează cu memoria, I/O etc.

Procesoarele x86 au 8 astfel de registre de 32 de biți. Deși oricare dintre acestea poate fi folosit în cadrul operațiilor, din motive istorice, fiecare registru are un rol anume.

Nume	Rol
EAX	acumulator; apeluri de sistem, I/O, aritmetică
EBX	registru de bază; folosit pentru adresarea bazată a memoriei
ECX	contor în cadrul instrucțiunilor de buclare
EDX	registru de date; I/O, aritmetică, valori de întrerupere; poate extinde EAX la 64 de biți
ESI	sursă în cadrul operațiilor pe stringuri
EDI	destinație în cadrul operațiilor pe stringuri
EBP	base sau frame pointer; indică spre cadrul curent al stivei
ESP	stack pointer; indică spre vârful stivei

Pe lângă acestea, mai există câteva registre speciale care nu pot fi accesate direct de către programator, cum ar fi EFLAGS și EIP (instruction pointer).

EIP este un registru în care se găsește adresa instrucțiunii curente, care urmează să fie executată. El nu poate fi modificat direct, programatic, ci indirect prin instrucțiuni de *jump*, *call* și *ret*.

Registrul EFLAGS conține 32 de biți folosiți pe post de indicatori de stare sau variabile de condiție. Spunem că un indicator/flag este setat dacă valoarea lui este 1. Cei folosiți de către programatori în mod uzual sunt următorii:

Nume	Nume extins	Descriere
CF	Carry Flag	Setat dacă rezultatul depășește valoarea întreagă maximă (sau minimă) reprezentabilă pe numere **unsigned**
PF	Parity Flag	Setat dacă byte-ul low al rezultatului conține un număr par de biți de 1
AF	Auxiliary Carry Flag	Folosit în aritmetică BCD; setat dacă bitul 3 generează carry sau borrow
ZF	Zero Flag	Setat dacă rezultatul instrucțiunii precedente este 0
SF	Sign Flag	Are aceeași valoare cu a bitului de semn din cadrul rezultatului (1 negativ, 0 pozitiv)
OF	Overflow Flag	Setat dacă rezultatul depășește valoarea întreagă maximă (sau minimă) reprezentabilă pe numere **signed**

Dacă urmăriți evoluția registrelor de la 8086, veți vedea că inițial se numeau AX, BX, CX etc. și aveau dimensiunea de 16 biți. De la 80386, Intel a extins aceste registre la 32 biți (i.e. "extended" AX → EAX).

Clase de instrucțiuni

Deși setul curent de instrucțiuni pentru procesoarele Intel are proporții biblice [http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf], noi ne vom ocupa de un subset [http://css.csail.mit.edu/6.858/2015/readings/i386.pdf] din acestea, și anume, o parte dintre instrucțiunile 80386.

Toate instrucțiunile procesoarelor x86 se pot încadra în 3 categorii: transfer de date, aritmetice/logice și de control. Vom enumera doar câteva instrucțiuni reprezentative, deoarece multe dintre ele se aseamănă.

Instrucțiuni de transfer de date

Nume	Operanzi	Descriere
mov	dst, src	Mută valoarea din sursă peste destinație
push	src	Mută valoarea din sursă în vârful stivei
pop	dst	Mută valoarea din vârful stivei în destinație
lea	dst, src	Încarcă adresa efectivă a sursei în destinație

xchg	dst, src	Interschimbă valorile din sursă și destinație

Instrucțiuni aritmetice și logice

Nume	Operanzi	Descriere
add	dst, src	Adună sursa cu destinația; rezultatul se scrie la destinație
sub	dst, src	Se scade din destinație sursa și se reține în destinație rezultatul
and	dst, src	Se efectuează operația de ȘI logic între sursă și destinație și se reține rezultatul în destinație
test	dst, src	Se efectuează operația de ȘI logic între sursă și destinație fără a se reține rezultatul undeva
shl	dst, <const>	Se face shiftare logică la stânga a destinației cu un număr constant de poziții

Instrucțiuni de control

Nume	Operanzi	Descriere
jmp	<adresă>	Efectuează salt necondiționat la adresa indicată (direct, prin registru, prin etichete)
cmp	dst, src	Compară sursa cu destinația (detalii mai jos)
j*condiție*	<adresă>	Efectuează salt condiționat, în funcție de valoarea flagului/variabilei de condiție
call	<adresă>	Face apel la subrutina care se găsește la adresa indicată

Instrucțiunea ''cmp dest, src'' [https://www.felixcloutier.com/x86/cmp] realizează în spate operația `dest - src` (adică scade din destinație sursa); este vorba de o scădere cu semn. Fără a reține rezultatul. Astfel, în cazul codului

```
    cmp eax, 0
    jl negative
```

se va face saltul la eticheta `negative` dacă eax este mai mic decât `0`. Se face operația `eax - 0` și dacă rezultatul este negativ (adică dacă eax este negativ) se face saltul.

Atunci când avem comparații cu `0` (*zero*), același lucru se poate face mai eficient folosind instrucțiunea `test`:

```
    test eax, eax
    jl negative
```

Alte detalii aici [https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow#Comparison_Instructions].

Exerciții

În cadrul laboratoarelor vom folosi repository-ul de git al materiei IOCLA - https://github.com/systems-cs-pub-ro/iocla [https://github.com/systems-cs-pub-ro/iocla]. Repository-ul este clonat pe desktop-ul mașinii virtuale. Pentru a îl actualiza, folosiți comanda `git pull origin master` din interiorul directorului în care se află repository-ul (`~/Desktop/iocla`). Recomandarea este să îl actualizați cât mai frecvent, înainte să începeți lucrul, pentru a vă asigura că aveți versiunea cea mai recentă.Dacă doriți să descărcați repository-ul în altă locație, folosiți comanda `git clone https://github.com/systems-cs-pub-ro/iocla [https://github.com/systems-cs-pub-ro/iocla] ${target}`.Pentru mai multe informații despre folosirea utilitarului `git`, urmați ghidul de la Git Immersion [https://gitimmersion.com/].

Pentru a afișa valorile din registre vom folosi macro-ul `PRINTF32` dezvoltat de Dragoș Niculescu până veți învăța cum se efectuează apelurile de funcții. Acesta permite afișarea de valori în diverse formate și de șiruri de caractere. Pentru mai multe detalii urmăriți descrierea din fișierul unde este definit macro-ul.

Pentru usurința în timpul dezvoltării, obișnuiți-vă să faceți referire la documentația instrucțiunilor - de transfer de date [https://en.wikibooks.org/wiki/X86_Assembly/Data_Transfer], aritmetice

[https://en.wikibooks.org/wiki/X86_Assembly/Arithmetic], de control [https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow].

0. Walkthrough

Accesați directorul `0-walkthrough` din arhiva laboratorului.

1. Deschideți fișierul `ex1.asm` și citiți comentariile. Asamblați folosind utilitarul make și rulați. Folosind gdb, parcurgeți programul linie cu linie (comanda `start` urmată de `next`) și observați schimbarea valorilor registrelor în urma executării instrucțiunilor `MOV` și `ADD`. Ignorați secvența de instrucțiuni a macro-ului `PRINTF32`.
2. Deschideți fișierul `ex2.asm` și citiți comentariile. Asamblați folosind utilitarul make și rulați. Folosind gdb, observați schimbarea registrului **EIP** la executarea instrucțiunii `JMP`. Pentru a trece peste instrucțiunile macro-ului `PRINTF32`, adăugați un breakpoint la label-ul `jump_incoming` (comanda `break` urmată de `run`).
3. Deschideți fișierul `ex3.asm` și citiți comentariile. Asamblați folosind utilitarul make și rulați. Folosind gdb, parcurgeți programul folosind breakpoint-uri. Urmăriți flow-ul programului. De ce se afișează mai întâi 15 și după aceea 3? Din cauza jump-ului de la linia 9. Către ce locație indică jump-ul de la linia 25? Către label-ul `zone1`.
4. Deschideți fișierul `ex4.asm` și citiți comentariile. Asamblați folosind utilitarul make și rulați. Folosind gdb, parcurgeți programul. De ce nu se ia jump-ul de la linia 12? Pentru că instrucțiunea `JE` face saltul dacă este setat bit-ul **ZF** din registrul **FLAGS**. Acesta este setat de instrucțiunea `CMP`, care face diferența dintre valorile registrelor **EAX** și **EBX** fără a stoca rezultatul. Însă `ADD`-ul de la linia 11 șterge acest flag, deoarece rezultatul operației este diferit de 0.

1. Conditional jumps

Accesați directorul `1-2-hello-world` din arhiva laboratorului. Modificați programul astfel încât afișarea mesajului să se facă numai dacă conținutul registrului **eax** este mai mare decât cel din **ebx**. Modificați și valoarea registrelor pentru a face în continuare afișarea mesajului `"Hello, World!"`.

2. More hellos

1. Modificați programul astfel încât să mai afișeze încă un mesaj ('Goodbye, World!')
2. Folosind instrucțiuni de tip jump, modificați programul astfel încât să afișeze de N ori 'Hello, World!', unde N este dat prin intermediul registrului ECX. Evitați ciclarea la infinit.

După rezolvarea cu succes, programul ar trebui să se afișeze:

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Goodbye, World!
```

3. Grumpy jumps

Accesați directorul `3-grumpy-jumps`. Treceți prin codul sursă din grumpy-jumps.asm.

1. Modificați valorile registrelor EAX si EBX astfel încât la rularea lui să se afișeze mesajul `Well done!`. Urmăriți comentariile marcate cu `TODO`
2. De ce, în continuare, se afișează și mesajul greșit? Modificați sursa astfel încât să nu se mai afișeze mesajul greșit.

Pentru a determina valorile necesare pentru registrele EAX si EBX vă recomandăm să folosiți GDB.

4. Sets

Pornind de la scheletul de cod din directorul `4-sets` va trebui să implementați operații pe mulțimi ce pot conține elemente între 0 și 31. Un mod eficient de a face asta (atât din punct de vedere al spațiului cât și al vitezei) ar fi să reprezentăm mulțimile astfel încât un registru să reprezinte o mulțime. Fiecare bit din registru va reprezenta un element din mulțime (dacă bit-ul i este setat atunci mulțimea conține elementul i)

Exemplu: dacă eax ar conține reprezentarea mulțimii `{0,2,4}`, valoarea registrului ar fi `2^0 + 2^2 + 2^4 = 1 + 4 + 16 = 21`. Documentați-vă despre instrucțiunile disponibile pe arhitectura x86 [http://www.cs.virginia.edu/~evans/cs216/guides/x86.html].

- Aveți definite 2 mulțimi. Ce valori conțin? Realizați reuniunea celor 2 mulțimi.

- Folosiți instrucțiunea `or` pentru a adăuga două elemente noi în mulțime.

Folosiți-vă de faptul că mulțimile curente, deși au "spațiu" pentru 32 de biți, au doar 8 biți folosiți. Dacă veți face `or` cu un număr mai mare de 255 (`0xff`, `2^8-1`) care are doi biți activi, veți adăuga practic două elemente noi la mulțime.

- Faceți intersecția celor 2 mulțimi.

- Determinați elementele care lipsesc din mulțimea `eax` pentru ca aceasta să fie completă.

Adică trebuie să faceți complementul numărului folosind instrucțiunea `not`.

- Eliminați un element din prima mulțime.

- Faceți diferența între mulțimi.

Pentru a vă ajuta în afișare puteți folosi macro-ul `PRINTF32`. De exemplu:

```
PRINTF32 `Reuniunea este: \x0`
PRINTF32 `%u\n\x0`, eax
```

5. BONUS: Min

Calculați minimul dintre numerele din 2 registre (eax și ebx) folosind o instrucțiune de salt și instrucțiunea `xchg`.

6. BONUS: Fibonacci

Calculați al `N`-lea număr Fibonacci, unde `N` este dat prin intermediul registrului `eax`.

Resurse utile

- Programming from the Ground Up [http://savannah.nongnu.org/projects/pgubook/]
- Reverse Engineering for Beginners [http://beginners.re/]

- Intel 64 and IA-32 Architectures Software Developer Manual [http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf]
- Intel 80386 Programmer's Reference Manual [http://css.csail.mit.edu/6.858/2015/readings/i386.pdf]
- RISC vs. CISC [http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/]
- Intel x86 JUMP Quick Reference [http://unixwiz.net/techtips/x86-jumps.html]

Soluții

Soluțiile pentru exerciții sunt disponibile aici [https://elf.cs.pub.ro/asm/res/laboratoare/lab-05-sol.zip].