Laborator 03

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:

```
10110000000011000110011000110001110111111111100100
```

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 sau OllyDbg, 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. 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 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.

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. Se spune 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	Acest flag este setat dacă rezultatul instrucțiunii precedente a generat carry sau borrow
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

Nume	Nume extins	Descriere
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

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, noi ne vom ocupa de un subset 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 sursă destinația și se reține în destinație
and	dst, src	Se efectuează operația de ȘI logic între sursă și destinație
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 (scade) sursa cu destinația, setând flag-urile în mod corespunzător
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

Exemple

Limbajul de asamblare x86 are două sintaxe oficiale: Intel și AT&T. Există o serie de diferențe între cele două. Sintaxa Intel este sprijinită de majoritatea asambloarelor. Din considerente de platformă și răspândire (și pentru că programatorii scriu a = 1, nu 1 = a), noi vom folosi sintaxa Intel în cadrul laboratorului.

Hello, World!

Putem vedea un exemplu de program în limbaj de asamblare mai jos. Acesta va afișa, la consolă, string-ul `Hello, World!`.

```
%include "io.inc"
extern puts

section .data
    myString: db 'Hello, World!',

section .text
global CMAIN
CMAIN:
    mov ebp, esp        ; Initialize frame pointer
    lea eax, [myString] ; Load the effective address of our string into the
eax register
    push eax            ; Push the address unto the stack in order to pass
it to 'puts'
    call puts           ; Call the puts routine
    pop eax             ; Retrieve eax from the stack
    ret                 ; Return from the main routine
```

JMP și JMP-if-condition

Fluxul programelor în limbaj de asamblare este controlat prin instrucțiuni de tip **jump**, un analog al lui **goto** din limbajul C.

Instrucțiunea **jmp** va dirija fluxul programului spre adresa primită ca argument, fie direct, fie printr-un registru.

Spre exemplu:

```
%include "io.inc"
extern puts

section .data
    string1: db "You'll never get here!",
    string2: db "Nothing to see here.",
```

```
section .text
global CMAIN
CMAIN:
    mov ebp, esp
    lea eax, [string2]
    push eax
    call puts
    pop eax
    jmp exit           ; Unconditional jump to the 'exit' label
    lea eax, [string1] ; Code unreachable beyond this point
    push eax
    call puts
    pop eax
exit:
    ret
```

Instrucțiunile de tipul **jump-if-condition** se aseamănă cu **if** din C. Aceste instrucțiuni folosesc drept condiții indicatorii de stare din registrul EFLAGS. E foarte important de ținut minte faptul că acest registru indică contextul de execuție al instrucțiunii curente și se modifică după fiecare instrucțiune.

Pentru a exemplifica acest lucru, fie următorul program:

```
%include "io.inc"
extern puts

section .data
    myString: db "Hello, World!",

section .text
global CMAIN
CMAIN:
    mov ebp, esp
    mov eax, 1
    mov ebx, 1
    cmp eax, ebx
    add ecx, 1   ; Delete this line
    je print
    ret
print:
    lea eax, [myString]
    push eax
    call puts
    pop eax
    ret
```

Observați ce se întâmplă atunci când ștergeți instrucțiunea indicată. Instrucțiunea **je print** va face jump doar dacă ZF este setat. Instrucțiunea **cmp** face diferența dintre cei doi operanzi; dacă diferența este 0, atunci ZF va fi setat. În schimb, instrucțiunea **add ecx, 1** modifică valoarea lui ZF. De aceea, se recomandă ca instrucțiunile de tipul j*condiție* să fie plasate imediat după instrucțiunile ce verifică condiția respectivă.

Exerciții

More hellos

1. Folosind programul ce afișează 'Hello, World!' de mai sus, încărcați adresa șirului în alt registru și puneți-l pe stivă. S-a schimbat ceva? De ce da/nu?
2. Modificați programul astfel încât să mai afișeze încă un mesaj ('Goodbye, World!')
3. Folosind instrucțiuni de tip jump, modificați programul astfel încât să afișeze de 3 ori 'Hello, World!'. Evitați ciclarea la infinit.

Grumpy jumps

Fie următorul program.

```
%include "io.inc"
extern puts

section .data
    wrong: db 'Not today, son.',
    right: db 'Well done!',

section .text
global CMAIN
CMAIN:
    mov ebp, esp        ; set the frame pointer
    mov eax, 0xdeadc0de ; TODO
    mov ebx, 0x1337ca5e ; TODO
    mov ecx, 0x5        ; hardcoded
    cmp eax, ebx
    jns bad
    cmp ecx, ebx
    jb bad
    add eax, ebx
    xor eax, ecx
    jnz bad
good:
    lea eax, [right]
    push eax
    call puts
    pop eax
bad:
    lea eax, [wrong]
    push eax
    call puts
    pop eax
    ret
```

1. Modificați-l 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? Ce lipsește?

Resurse utile

- Intel 64 and IA-32 Architectures Software Developer Manual
- Intel 80386 Programmer's Reference Manual
- RISC vs. CISC