x86 assembly – úvod

Milan Radojčić

SSPŠaG – KBB

28. dubna 2026

Hlavní body

Hlavní body

Motivace

K čemu se učit assembly?

- reverzní inženýrství
- analýza malware
- psaní exploitů (shellcode)
- zvědavost!

Historie architektury x86

- Originálně instrukční sada pro procesor Intel 80**86** a později pro procesory 801**86**, 802**86**, atd.
- Sada byla originálně 16-bitová, později rozšířena na 32 a 64 bitů.
  - Tomuto 32-bitovému rozšíření se někdy říka i386.
  - 64-bitová sada se podobně někdy nazývá amd64 nebo x64.

Hlavní body

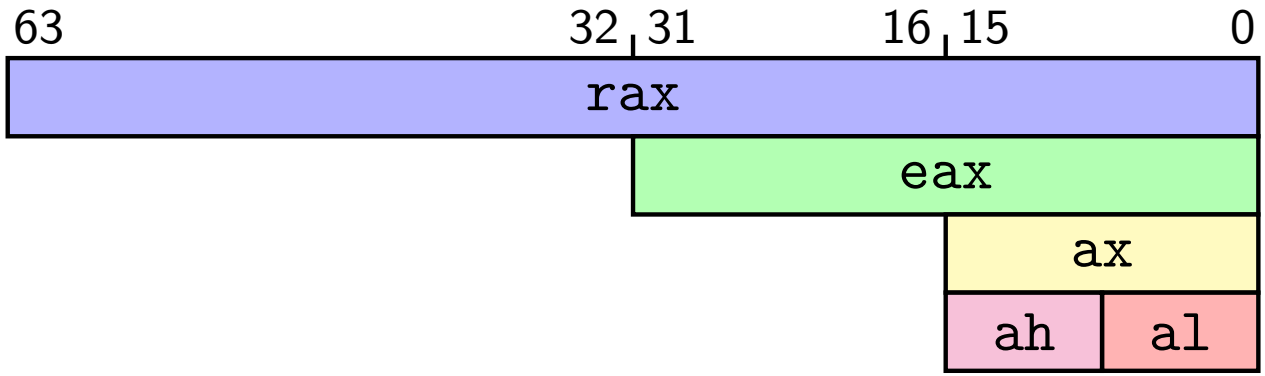
Registry

- Registr - malá volatilní paměť přímo v procesoru
- Originální 16-bitové registry:
  - `ax` - akumulátor (výstup z ALU)
  - `bx` - base register
  - `cx` - count register
  - `dx` - data register
  - `si` - source index
  - `di` - destination index
  - `bp` - base pointer
  - `sp` - stack pointer

Registry

- Když před název registru dáme `e` získáme tím 32-bitový registr (např. `eax`), popřípadě `r` pro 64-bitový registr (`rax`).
- Další 64-bitové registry: `r8` - `r15`
- Kromě těchto obecných registrů obsahuje architektura x86 i speciální registry pro čísla s pohyblivou desetinnou čárka.

Registry



Registry

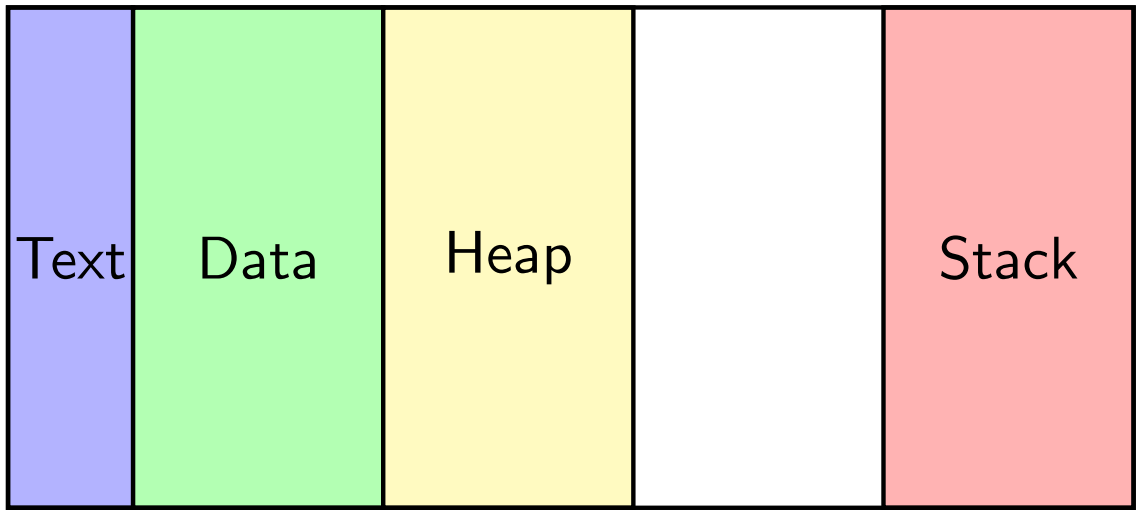
- `rax` – celý 64-bitový registr
- `eax` – spodních 32 bitů
- `ax` – spodních 16 bitů
  - `al` – spodní byte z `ax`
  - `ah` – horní byte z `ax`

První instrukce - mov

Pomocí této instrukce můžeme zapsat konstantu do registru.

```
mov rax, 6 ; zapíše do registru rax číslo 6
mov eax, 123
mov cx, 0x10
```

Rozdělení paměti



Pomocí příkazu `cat /proc/$$/maps` se můžeme podívat, jak jsou namapované různé sekce paměti spuštěného shellu.

Definování statických proměnných

```
    segment .data
a dq 175 ; statická proměnná s názvem "a"
b dq 4097 ; statická proměnná s názvem "b"
```

Nahrávání proměnných z paměti

```
    segment .data
a dq 175 ; dq - define quadword - 64 bitů
b dd 0xffffffff ; dd - define doubleword - 32 bitů
    segment .text
    global main
main:
    mov rax, a ; nahraje ADRESU a do rax
    mov rbx, [a] ; nahraje HODNOTU a do rbx
    mov rcx, [b]
```

Nahrávání proměnných z paměti

```
    segment .data
a db 1, 2, 3, 4 ; db - define bytes - array bytů
    segment .text
    global main
main:
    mov bl, [a] ; nahraje PRVNI hodnotu a do rbx
    mov cl, [a + 1] ; nahraje druhou hodnotu
    mov dl, [a + 2]
```

Nahrávání proměnných z paměti

```
    segment .data
a dw 1, 2, 3, 4 ; db - define word - array dvoubytu
    segment .text
    global main
main:
    mov rbx, [a] ; nahraje PRVNI hodnotu a do rbx
    mov rcx, [a + 2 * 1] ; nahraje druhou hodnotu
    mov edx, [a + 2 * 2]
```

Nahrávání proměnných z paměti – přímé adresování

```
    section .data
mojepromenna dq 321
    section .text
    global main
main:
    mov rcx, [mojepromenna]
    nop
```

Nahrávání proměnných z paměti – přímé adresování

```
(gdb) disassemble main
Dump of assembler code for function main:
 0x0000000000401110 <+0>:     mov    0x404010,%rcx
 0x0000000000401118 <+8>:     nop
```

Nahrávání proměnných z paměti – relativní adresování

```
    section .data
mojepromenna dq 321
    section .text
    global main
main:
    mov rcx, [rel mojepromenna]
    nop
```

Nahrávání proměnných z paměti – relativní adresování

```
(gdb) disassemble main
Dump of assembler code for function main:
 0x0000000000001120 <+0>:     mov    0x2ee9(%rip),%rcx
 0x0000000000001127 <+7>:     nop
```

Nahrávání proměnných z paměti – relativní adresování

```
    default rel ; macro pro nasm

    section .data
mojepromenna dq 321
    section .text
    global main
main:
    mov rcx, [mojepromenna]
    nop
```

Další použití `mov`

- Přesunutí z registru do paměti.

```
mov [a], rax
```

- Přesunutí z registru do registru.

```
mov rbx, rax
```

Hlavní body

GDB

- Zkratka pro **G**NU **D**e**b**ugger
- Low-level debugger, který umí:
  - disassemblovat
  - nastavovat breakpointy
  - stepovat po instrukcích
  - vypisovat stav registrů a paměti
- Spustíme jej s naším programem takto: `gdb [program]`.

Disassemblování v GDB

```
(gdb) disassemble main
Dump of assembler code for function main:
    0x0000000000001120 <+0>:    mov    $0x1,%eax
    0x0000000000001125 <+5>:    mov    $0x2,%ebx
    0x000000000000112a <+10>:   mov    $0x3,%ecx
    0x000000000000112f <+15>:   mov    $0x7b,%r8d
    0x0000000000001135 <+21>:   mov    $0x10,%edi
    0x000000000000113a <+26>:   nop
    0x000000000000113b <+27>:   mov    $0x7b,%eax
End of assembler dump.
(gdb)
```

Nastavení breakpointu v GDB

```
(gdb) break *0x000000000000113a
Breakpoint 1 at 0x113a
(gdb) b *main+26
Note: breakpoint 1 also set at pc 0x113a.
Breakpoint 2 at 0x113a
(gdb) delete 2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000000113a <main+26>
(gdb)
```

Spouštění programu v GDB

```
(gdb) run
Starting program: ...
Breakpoint 1, 0x00005555555513a in main ()
(gdb) disassemble main
Dump of assembler code for function main:
    0x0000555555555120 <+0>: mov    $0x1,%eax
    0x0000555555555125 <+5>: mov    $0x2,%ebx
    0x000055555555512a <+10>: mov    $0x3,%ecx
    0x000055555555512f <+15>: mov    $0x7b,%r8d
    0x0000555555555135 <+21>: mov    $0x10,%edi
 => 0x000055555555513a <+26>: nop
    0x000055555555513b <+27>: mov    $0x7b,%eax
End of assembler dump.
(gdb)
```

Vypisování hodnot registrů

```
(gdb) info register rax
rax    0x1    1
(gdb) i r rbx
rbx    0x2    2
(gdb) i r rcx
rcx    0x3    3
(gdb) i r r8
r8     0x7b   123
(gdb) i r edi
edi    0x10    16
(gdb) i r rdx
rdx    0x7fffffffdcb8    140737488346296
```

Vypisování paměti

```
    section .data
a dq 123
str: db "ahoj", 0
    section .text
    global main
main:
    nop
```

Vypisování paměti

```
(gdb) x/dg &a
0x555555558010 <a>: 123
(gdb) x/s &str
0x555555558018 <str>: "ahoj"
(gdb)
```

Specifikace formátu pro výpis přes `x`

Formát za lomítkem je ve tvaru: `NFS` (počet, formát a velikost). Možné velikosti a formáty jsou:

Písmeno	Počet bytů
b	1
h	2
w	4
g	8

Písmeno	Formát
d	decimal
x	hexadecimal
t	binary
u	unsigned
f	floating point
i	instruction
c	character
s	string
a	address

Shrnutí příkazů v GDB

- **nejdůžitější** – `help`
- `disassemble [funkce]`
- `break [adresa]`, `b` – nastaví breakpoint na konkrétní adresu
- `delete`, `d` – smaže breakpoint
- `info breakpoints`, `i b` – vypíše informace o breakpointech
- `run` – spustí program
- `info register [registr]`, `i r` – vypíše hodnotu v registru (bez argumentu vypíše hodnoty všech registrů)
- `x/[formát] [adresa]` – vypíše hodnotu z paměti

Hlavní body

Instrukce add

- K prvnímu argumentu přičítá druhý argument.
- Nastavuje flagy v registru `rflags`:
  - overflow (`OF`) – když dojde k přetečení
  - sign (`SF`) – nastaveno na znaménko výsledku
  - zero (`ZF`) – nastaveno na 1 pokud je výsledek je 0

Instrukce add

```
    segment .text
    global main
main:
    mov rax, 2
    add rax, 1 ; přičte 1 k rax
    mov rbx, 3
    add rax, rbx ; přičte rbx k rax
```

Instrukce add

```
    segment .data
a dq 123
b dq 321
    segment .text
    global main
main:
    mov rax, [a]
    add rax, [b] ; přičte hodnotu v b k rax
    add [a], 321 ; přičte 321 k a
```

Instrukce `sub`

- Od prvního argumentu odečte druhý argument.
- Nastavuje flagy v registru `rflags`:
  - overflow (`OF`) – když dojde k přetečení
  - sign (`SF`) – nastaveno na znaménko výsledku
  - zero (`ZF`) – nastaveno na 1 pokud je výsledek je 0

Instrukce sub

```
    segment .text
    global main
main:
    mov rax, 2
    sub rax, 1 ; odečte 1 od rax
    mov rbx, 3
    sub rax, rbx ; odečte rbx od rax
```

Instrukce sub

```
    segment .data
a dq 123
b dq 321
    segment .text
    global main
main:
    mov rax, [b]
    add rax, [a] ; odečte hodnotu v a od rax
    add [a], 23 ; odečte 23 od a
```

Instrukce `cmp`

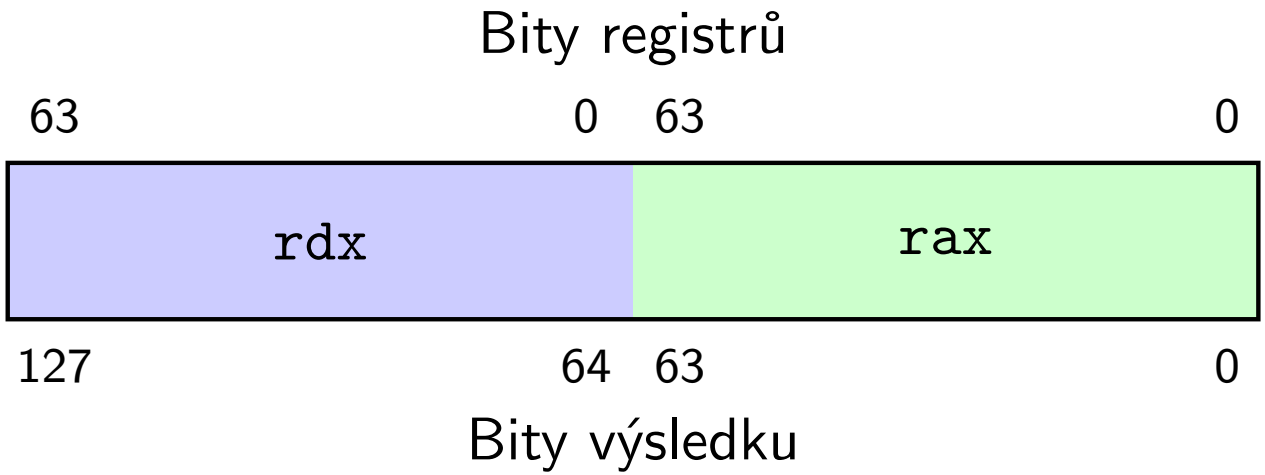
- Odečte od prvního argumentu druhý argument a podle toho nastaví flagy, výsledek neukládá.
- Užitečná bude až si později budeme povídat o podmínených skocích.

Instrukce `mul` a `imul`

- `mul` – provádí unsigned násobení
- `imul` – provádí signed násobení
- Můžou brát 1, 2 nebo 3 parametry.

Instrukce `mul` a `imul` – 1 parametr

- Vynásobí předaný parametr (registr nebo adresa) hodnotou v `rax` a výsledek uloží do `rdx:rax`.
  - Nižší bity výsledku jsou uloženy v registru `rax` a vyšší v registru `rdx`.



Instrukce `mul` a `imul` – 1 parametr

```
mov rax, 2
mov rbx, 3
imul rbx
```

Po spuštění tohoto kódu bude v registru `rax` uložena hodnota 6.

Instrukce `mul` a `imul` – 2 parametry

```
imul rax, 100 ; uloží do rax hodnotu rax * 100
imul r8, [x] ; uloží do r8 hodnotu r8 * x
imul r9, r10 ; uloží do r9 hodnotu r9 * r10
```

Instrukce `mul` a `imul` – 3 parametry

```
imul rbx, [x], 100 ; uloží do rbx hodnotu x * 100
imul rdx, rbx, 50 ; uloží do rdx hodnotu rbx * 50
```

Instrukce `mul` a `imul`

- `mul` a `imul` nastavují carry a overflow flagy když výsledek překročí 64 bitů.
- Jiné flagy (např. zero flag atd.) nastaveny nejsou.

Instrukce `div` a `idiv`

- Instrukce `div` bere jenom jeden parametr, kterým je dělitel. (registr nebo adresa)
- V registrech `rdx:rax` má být uložen dělenec.
- Po provedení instrukce bude v registru `rax` uložený výsledek a v registru `rdx` uložen zbytek.
  - Trochu funguje jako opak instrukce `mul`.
- Nenastavuje žádné flagy.

Instrukce div a idiv

```
mov rax, [x]
mov rdx, 0 ; vynulování horních bitů rdx:rax
idiv [y] ; provede rdx:rax / y
mov [quot], rax ; uloží výsledek do quot
mov [rem], rdx ; uloží zbytek do rem
```