Einführung in Assembly

x86, 64-bit

Jan-Niklas

2023-03-09

Maschinencode und Assembly

- Maschinencode: Code den die CPU direkt ausführt
- Assembly: Maschinencode in lesbarer Form
- Spezifisch für jede CPU-Architektur
    - Hier nur x86, 64-bit

Warum?

- Reverse Engineering: Programm als Maschinencode gegeben
  - Maschinencode verstehen
- Shellcoding: Kleine handgeschriebene Codeschnipsel
  - Maschinencode schreiben
- Low-level Interaktion mit CPU
- High-Performance Code, Performance-Analyse

Syntax

- Programm: Folge von Instruktionen
  - Keine high-level Konstrukte
- Intel oder AT&T Syntax
  - Wir betrachten nur Intel
- Instruktion: Mnemonic und Operanden
  - Zieloperand zuerst
  - `mov rax, rbx` entspricht `rax = rbx`
- Mögliche Operanden:
  - Register: Speicherstellen direkt in der CPU
    - `rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15`
  - Immediates: Zahlen kodiert in der Instruktion
    - `add rax, 5`
  - Memory: Speicher an gegebener Adresse
    - `add rax, qword ptr [rbx + rsi * 8]`

Instruktionen: ALU

- Arithmetisch/logisch, zwei Operanden:
  - `add, sub, and, or, xor, shl, shr`
- Arithmetisch/logisch, ein Operand:
  - `inc, dec, neg, not`
- Multiplikation und Division:
  - `mul, imul, div, idiv`: Ein Operand
  - `mul op` entspricht `rdx:rax = rax * op`
  - `imul` auch mit zwei Operanden
- Adressberechnung:
  - `lea`
- Sonstiges:
  - `mov, nop`

Operanden: Register

- Register sind 64-bit, können auch als 32-bit, 16-bit, oder 8-bit adressiert werden
- 64-bit: `rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15`
- 32-bit: `eax, ebx, ecx, edx, esi, edi, ebp, esp, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d`
- 16-bit: `ax, bx, cx, dx, si, di, bp, sp, r8w, r9w, r10w, r11w, r12w, r13w, r14w, r15w`
- 8-bit: `al, bl, cl, dl, sil, dil, bpl, spl, r8b, r9b, r10b, r11b, r12b, r13b, r14b, r15b`

Operanden: Speicher

- Adresse aus Base (register), Index (register), Scale (immediate), Displacement (immediate)
- `mov rax, qword ptr [rax + rbx * 8 + 1024]`
- `qword`: 64-bit, `dword`: 32-bit, `word`: 16-bit, `byte`: 8-bit
- Größe kann weggelassen werden wenn eindeutig
- `lea`: Berechnet Adresse ohne Speicherzugriff
  - `lea rax, [rax + rbx * 8 + 1024]`
  - `rax = rax + rbx * 8 + 1024`

Beispiele

```
add rax, 5

or ecx, dword ptr [rax]

not qword ptr [rbx + rdi]

imul rbx, qword ptr [rsp - 32]

lea rdx, [rdx + 2 * rax]

mov [rax], eax
```

Kontrollfluss

- Keine high-level Konstrukte wie `if, while, for`
- Jump: `jmp` mit Label als Operand
- Label: Markiert Instruktion, gefolgt von `:` in Definition

```
loop:
  jmp loop
```

- Vom Assembler übersetzt in relativen Sprung

Kontrollfluss

- Bedingungen: Vergleich gefolgt von bedingtem Sprung
- Vergleich mit `cmp` Instruktion (zwei Operanden)
  - `cmp rax, 5`
- Bedingter Sprung:
  - `je, ja, jb, jl, jg`: Jump if equal/above/below/less/greater
  - Above/below: unsigned, less/greater: signed
  - `e` anhängen für “or equal”: `jae, jbe, jle, jge`
  - `n` davor für “not”: `jne, jna, jnae, jnb, jnbe, ...`

Kontrollfluss

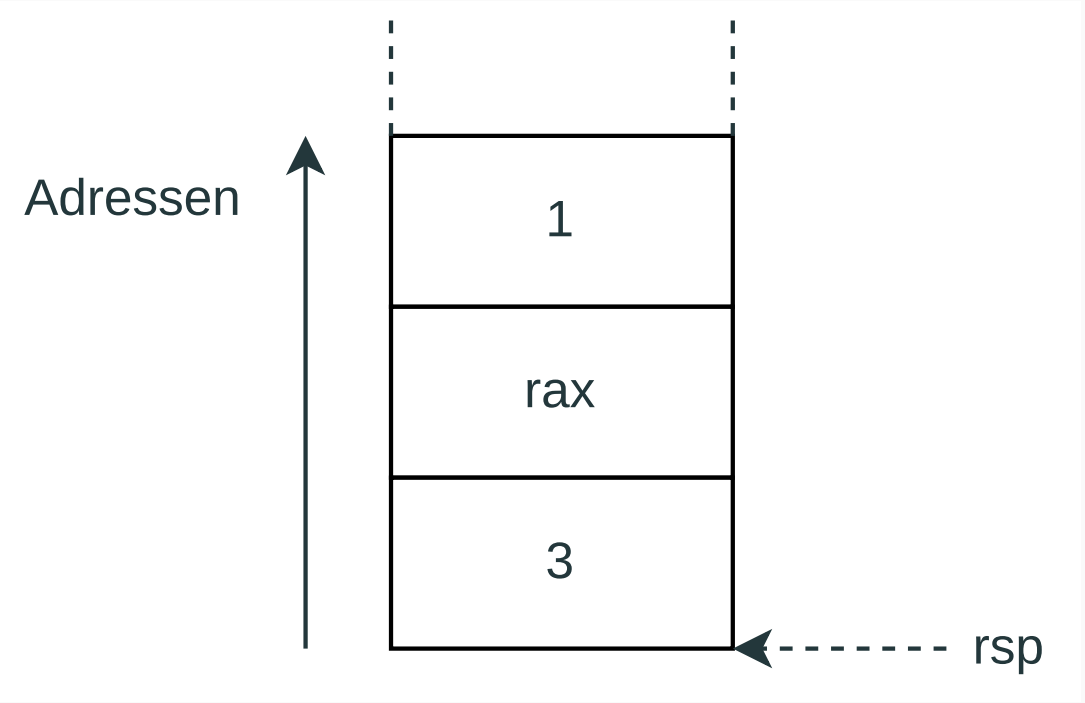
```
    ; rax = 0
    mov rax, 0
    ; rcx = 0
    mov rcx, 0

loop:
    ; rax += rcx
    add rax, rcx
    ; rcx++
    inc rcx
    ; Wiederhole wenn rcx < 5
    cmp rcx, 5
    jb loop
```

Stack

- Bereich des Hauptspeichers
- Temporärer Speicher, falls Register nicht reichen
- Instruktionen: `push`, `pop`, ein Operand
  - `push 5`
  - `pop rax`
- Register `rsp` zeigt auf “oberstes” Element des Stacks
- Stack wächst “nach unten”, zu niedrigeren Adressen
  - `rsp` zeigt auf niedrigste Adresse des Stack
  - Platz reservieren: `sub rsp, 64`
  - Platz freigeben: `add rsp, 64`

Stack

```
push 1
push rax
push 3
```



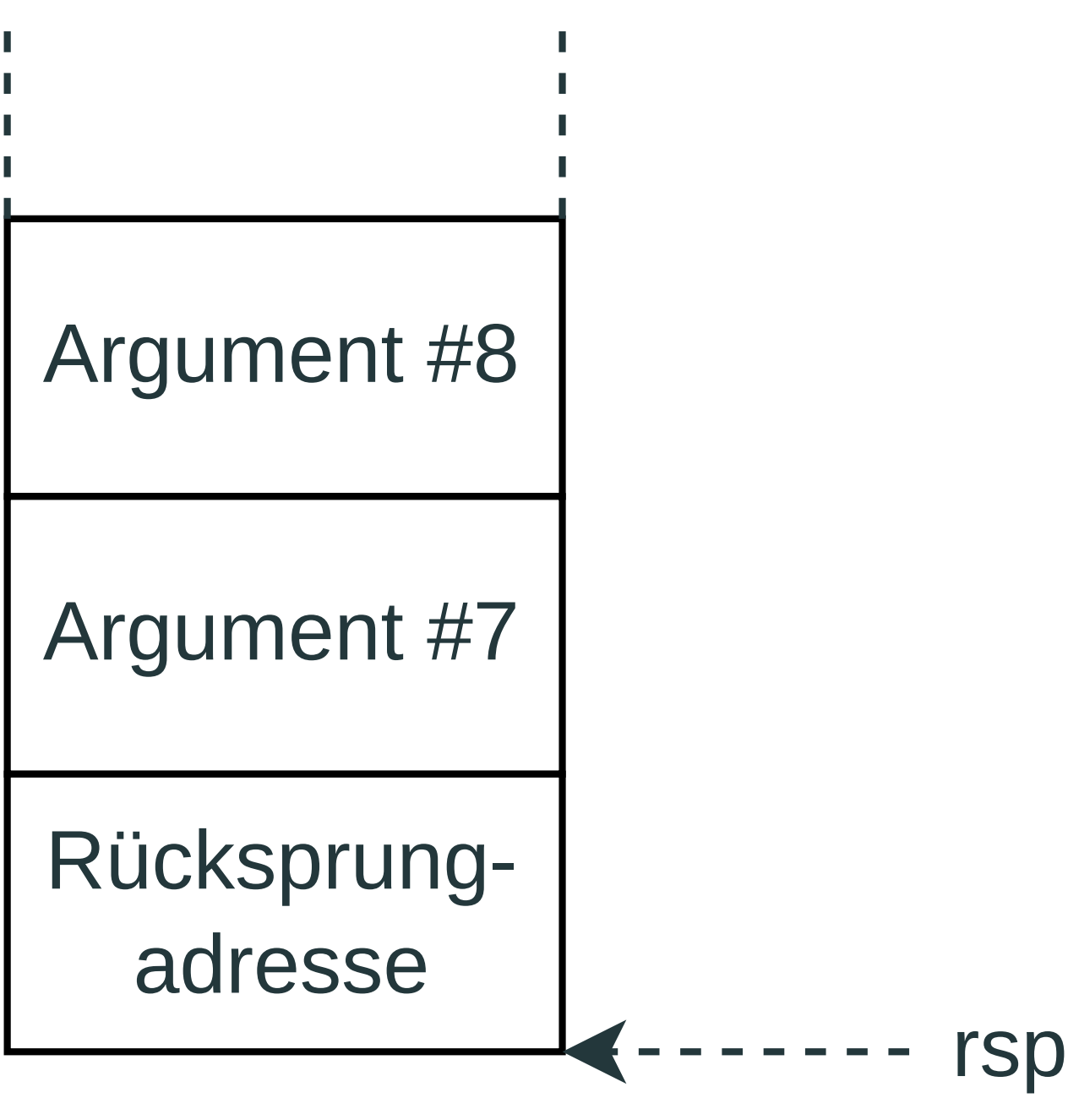
Funktionen

- Segmente von Code, von anderem Code aufrufbar
- Argumente übergeben, Rückgabewert erhalten
- Funktion aufrufen: `call label`
    - Speichert Adresse der folgenden Instruktion (Rücksprungadresse) auf dem Stack
    - Springt zur Funktion
- Zurückspringen: `ret`
    - Entfernt Rücksprungadresse vom Stack
    - Springt dorthin

Funktionen: Calling Convention

- Argumente in Registern:
  - `rdi, rsi, rdx, rcx, r8, r9`
  - Weitere Argumente auf dem Stack
- Rückgabewert in `rax`
- Funktion darf Register ändern (caller-saved):
  - `rax, rcx, rdx, rsi, rdi, r8-r11`
- Funktion muss Register wiederherstellen (callee-saved):
  - `rbx, rsp, rbp, r12-r15`
- Stack muss auf 16-byte aligned sein

Funktionen: Calling Convention



Funktionen

```
# Addiere 1, 2, 3
mov rdi, 1
mov rsi, 2
mov rdx, 3
call add3
# Addiere 4, 5 dazu
mov rdi, rax
mov rsi, 4
mov rdx, 5
call add3


# Addiert drei Argumente zusammen
add3:
  mov rax, rdi
  add rax, rsi
  add rax, rdx
  ret
```

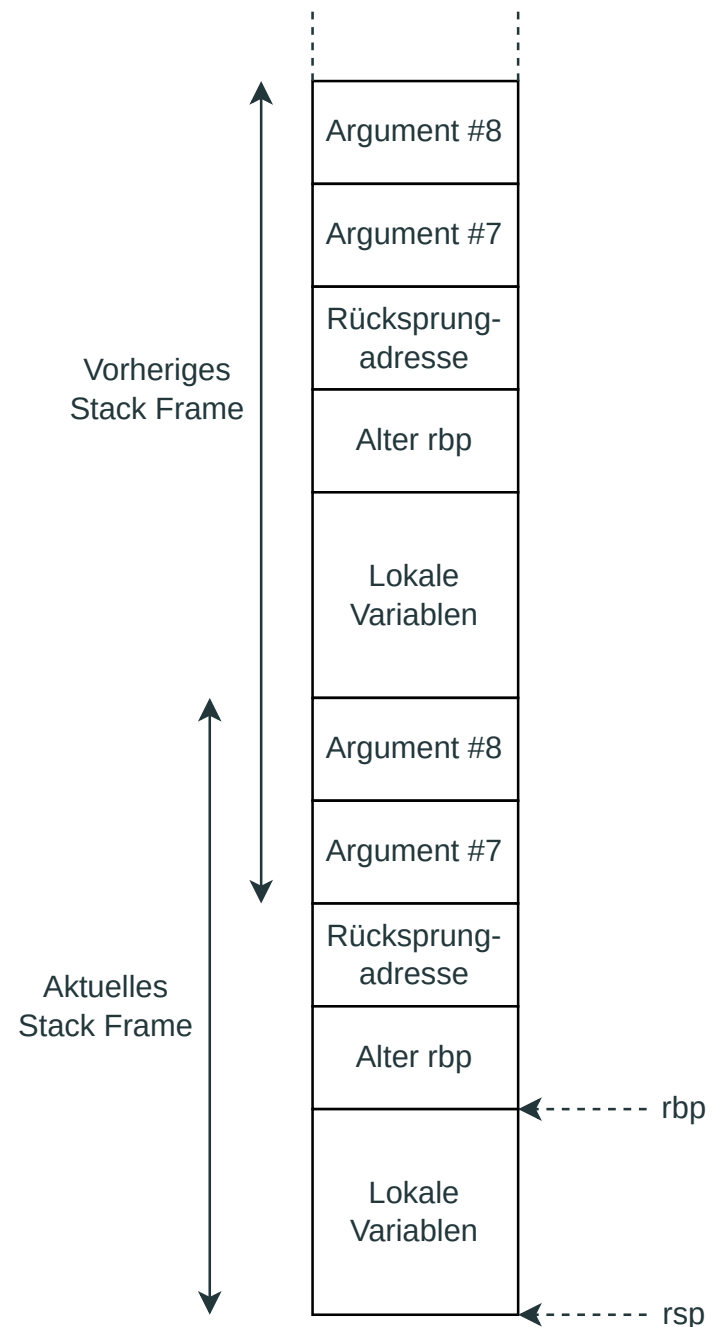
Funktionen: Stack Frames

- Jeder Funktionsaufruf kann Stack benutzen, wird “pausiert” bei weiterem Funktionsaufruf
- Stack Frame: Bereich des Stacks der einem Funktionsaufruf “gehört”
- Üblicherweise wird `rbp` genutzt um sich den Anfang des eigenen Stack Frames zu merken
    - Kann auch als normales Register benutzt werden

Funktionen: Stack Frames

- Am Anfang der Funktion (Prolog):
  - `push rbp`: Wert von `rbp` sichern
  - `mov rbp, rsp`: Anfang des eigenen Stack Frames merken
  - `sub rsp, 64`: Platz auf dem Stack reservieren
- Am Ende der Funktion (Epilog):
  - `mov rsp, rbp`: Platz auf dem Stack wieder freigeben
  - `pop rbp`: Wert von `rbp` wiederherstellen
  - `ret`: Zur oberen Funktion zurückspringen

Funktionen: Stack Frames



C Code

- C Code wird vom Compiler in Assembly übersetzt
- Compiler Explorer: Assembly Code von verschiedenen Compilern
- Beispiel: https://godbolt.org/z/3EPb8rqs8

Links

- Referenz zu allen Instruktionen
  - https://www.felixcloutier.com/x86/
- Liste von allen Instruktionen mit Operanden
  - https://ref.x86asm.net/coder64.html
- Calling Convention
  - https://en.wikipedia.org/wiki/X86_calling_conventions
- Compiler Explorer
  - https://godbolt.org/
- Intel Manuals, z.B. 5060 Seiten “Combined Volumes”
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- x86 Instruction Encoding
  - https://wiki.osdev.org/X86-64_Instruction_Encoding

Challenges

- Programmieraufgaben
  - Funktion, die Fibonacci-Folge iterativ berechnet
  - Funktion, die Fibonacci-Folge rekursiv berechnet
  - Template: https://godbolt.org/z/srdv7G7ef
- Reversing
  - Drei Challenges
  - Gegeben Assembly von Funktion

    ```
    bool is_flag_correct(const char *flag)
    ```

Reversing 1

```
bool is_flag_correct(const char *flag):
     0x1119  ba00000000     mov edx, 0
  ┌─> 0x111e  83fa16         cmp edx, 0x16
┌──< 0x1121  7f19           jg 0x113c
│└   0x1123  4863c2         movsxd rax, edx
│└   0x1126  488d0dd70e00.  lea rcx, [rip + 0xed7] ; 0x2004 ; "flag{f1rst_one_1s_e4sy}"
│└   0x112d  0fb63401       movzx esi, byte [rcx + rax]
│└   0x1131  40383407       cmp byte [rdi + rax], sil   ; arg1
┌───< 0x1135  750b           jne 0x1142
│└└  0x1137  83c201         add edx, 1
│└└─< 0x113a  ebe2           jmp 0x111e
└└──> 0x113c  b801000000     mov eax, 1
 └    0x1141  c3             ret
 └──> 0x1142  b800000000     mov eax, 0
      0x1147  c3             ret
```

Reversing 2

```
bool is_flag_correct(const char *flag):
      0x1119  ba00000000      mov edx, 0
   ┌─> 0x111e  83fa18          cmp edx, 0x18
  ┌──< 0x1121  7f21            jg 0x1144
  │┆   0x1123  4863ca          movsxd rcx, edx
  │┆   0x1126  0fbe040f        movsx eax, byte [rdi + rcx] ; arg1
  │┆   0x112a  83e830          sub eax, 0x30
  │┆   0x112d  83f070          xor eax, 0x70
  │┆   0x1130  488d35cd0e00.   lea rsi, [rip + 0xecd] ; 0x2004 ; "FLAG;34t4qC_35B34q454qpN="
  │┆   0x1137  0fbe0c0e        movsx ecx, byte [rsi + rcx]
  │┆   0x113b  39c8            cmp eax, ecx
 ┌───< 0x113d  750b            jne 0x114a
 ┆│┆   0x113f  83c201          add edx, 1
 ┆│└─< 0x1142  ebda            jmp 0x111e
 ┆└──> 0x1144  b801000000      mov eax, 1
 ┆      0x1149  c3              ret
 └────> 0x114a  b800000000      mov eax, 0
        0x114f  c3              ret
```

Reversing 3

```
bool is_flag_correct(const char *flag):
      0x1119  ba00000000      mov edx, 0
   ┌─> 0x111e  83fa1b          cmp edx, 0x1b
  ┌──< 0x1121  7f2d            jg 0x1150
  │┌─  0x1123  4863f2          movsxd rsi, edx
  ││   0x1126  0fbe0437        movsx eax, byte [rdi + rsi] ; arg1
  ││   0x112a  8d4c10c6        lea ecx, [rax + rdx - 0x3a]
  ││   0x112e  89d0            mov eax, edx
  ││   0x1130  c1e81f          shr eax, 0x1f
  ││   0x1133  01d0            add eax, edx
  ││   0x1135  d1f8            sar eax, 1
  ││   0x1137  83c072          add eax, 0x72
  ││   0x113a  31c8            xor eax, ecx
  ││   0x113c  488d0dc10e00.   lea rcx, [rip + 0xec1] ; 0x2004 ; "^AZC1t0NtJvCIMq1svLr4Fp6ml'!"
  ││   0x1143  0fbe0c31        movsx ecx, byte [rcx + rsi]
  ││   0x1147  39c8            cmp eax, ecx
 ┌───< 0x1149  750b            jne 0x1156
 │└─┌  0x114b  83c201          add edx, 1
 │  └< 0x114e  ebce            jmp 0x111e
 │  └─> 0x1150  b801000000     mov eax, 1
 │      0x1155  c3             ret
 └────> 0x1156  b800000000     mov eax, 0
        0x115b  c3             ret
```