Übung zu Betriebssysteme

x64-Assembler im Detail

KW 50 / 2020

Andreas Ziegler

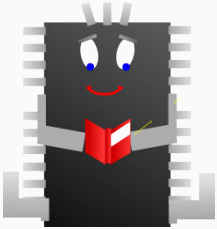
Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

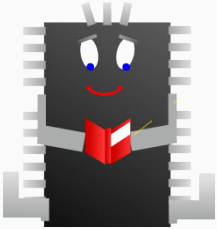
Mensch vs. CPU

```
48 83 ec 08
be 20 41 02 01
bf 20 45 02 01
e8 fd b9 ff ff
31 d2
be 23 00 00 00
bf 84 45 02 01
e8 8c c4 ff ff
48 8b 35 85 12 00 00
bf 20 45 02 01
e8 7b f1 ff ff
be 48 c7 00 01
48 89 c7
e8 6e f1 ff ff
be b0 b6 00 01
48 89 c7
e8 d1 f4 ff ff
e8 6c d1 ff ff
bf 20 40 02 01
e8 f2 af ff ff
bf a0 40 02 01
e8 88 b8 ff ff
e8 73 c1 ff ff
e8 0e 7d ff ff
fb
90
[...]
```



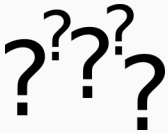
Mensch vs. CPU

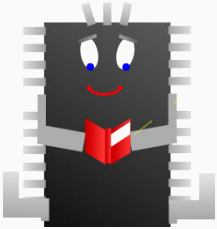
```
48 83 ec 08
be 20 41 02 01
bf 20 45 02 01
e8 fd b9 ff ff
31 d2
be 23 00 00 00
bf 84 45 02 01
e8 8c c4 ff ff
48 8b 35 85 12 00 00
bf 20 45 02 01
e8 7b f1 ff ff
be 48 c7 00 01
48 89 c7
e8 6e f1 ff ff
be b0 b6 00 01
48 89 c7
e8 d1 f4 ff ff
e8 6c d1 ff ff
bf 20 40 02 01
e8 f2 af ff ff
bf a0 40 02 01
e8 88 b8 ff ff
e8 73 c1 ff ff
e8 0e 7d ff ff
fb
90
[...]
```



Mensch vs. CPU

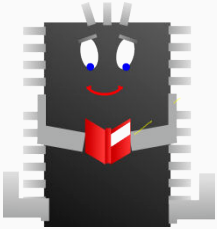
```
48 83 ec 08
be 20 41 02 01
bf 20 45 02 01
e8 fd b9 ff ff
31 d2
be 23 00 00 00
bf 84 45 02 01
e8 8c c4 ff ff
48 8b 35 85 12 00 00
bf 20 45 02 01
e8 7b f1 ff ff
be 48 c7 00 01
48 89 c7
e8 6e f1 ff ff
be b0 b6 00 01
48 89 c7
e8 d1 f4 ff ff
e8 6c d1 ff ff
bf 20 40 02 01
e8 f2 af ff ff
bf a0 40 02 01
e8 88 b8 ff ff
e8 73 c1 ff ff
e8 0e 7d ff ff
fb
90
[...]
```





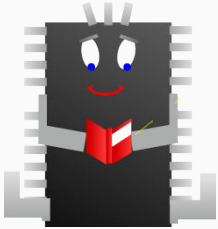
Mensch vs. CPU

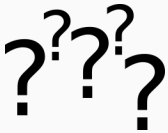
```
extern "C" int main() {
  TextStream::arrange(kout, dout);

  kout.setPos(35, 0);
  kout << os_name
     << "␣(3)" << endl;

  IOAPIC::init();

  keyboard.plugin();
  mouse.plugin();

  unsigned int numCPUs = Core::count();
  DBG_VERBOSE << "Number␣of␣CPUs:␣" << numCPUs << endl;

  ApplicationProcessor::boot();

  Core::Interrupt::enable();

  apps[Core::getID()].action();

  kapp.action();

  return 0;
}
```



Mensch vs. CPU

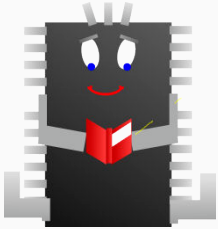
```
extern "C" int main() {
  TextStream::arrange(kout, dout);

  kout.setPos(35, 0);
  kout << os_name
     << "␣(3)" << endl;

  IOAPIC::init();

  keyboard.plugin();
  mouse.plugin();

  unsigned int numCPUs = Core::count();
  DBG_VERBOSE << "Number␣of␣CPUs:␣" << numCPUs << endl;

  ApplicationProcessor::boot();

  Core::Interrupt::enable();

  apps[Core::getID()].action();

  kapp.action();

  return 0;
}
```





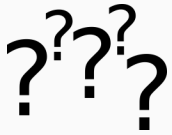
Mensch vs. CPU

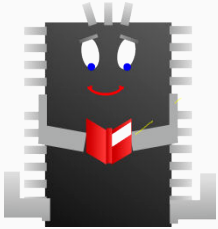
```
extern "C" int main() {
  TextStream::arrange(kout, dout);

  kout.setPos(35, 0);
  kout << os_name
     << "␣(3)" << endl;

  IOAPIC::init();

  keyboard.plugin();
  mouse.plugin();

  unsigned int numCPUs = Core::count();
  DBG_VERBOSE << "Number␣of␣CPUs:␣" << numCPUs << endl;

  ApplicationProcessor::boot();

  Core::Interrupt::enable();

  apps[Core::getID()].action();

  kapp.action();

  return 0;
}
```



Mensch vs. CPU

```
extern "C" int main() {
  TextStream::arrange(kout, dout);

  kout.setPos(35, 0);
  kout << os_name
     << "␣(3)" << endl;

  IOAPIC::init();

  keyboard.plugin();
  mouse.plugin();

  unsigned int numCPUs = Core::count();
  DBG_VERBOSE << "Number␣of␣CPUs:␣" << numCPUs << endl;

  ApplicationProcessor::boot();

  Core::Interrupt::enable();

  apps[Core::getID()].action();

  kapp.action();

  return 0;
}
```



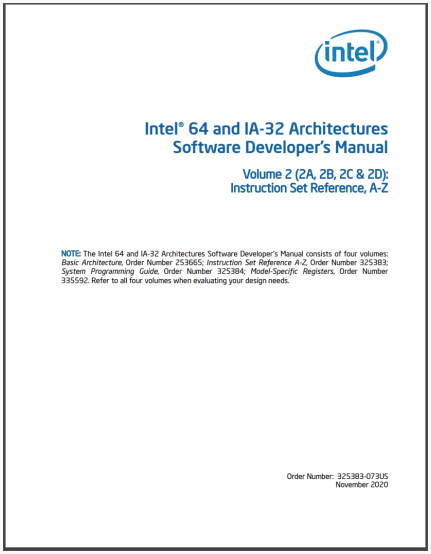


Kompromiss: Assembler Language

```
sub    $0x8,%rsp
mov    $0x1024120,%esi
mov    $0x1024520,%edi
callq  1007b80 <_ZN10TextStream7arrangeERS_PS_>
xor    %edx,%edx
mov    $0x23,%esi
mov    $0x1024584,%edi
callq  1008620 <_ZN10TextWindow6setPosEii>
mov    0x1285(%rip),%rsi
mov    $0x1024520,%edi
callq  100b320 <_ZN12OutputStreamlsEPKc>
mov    $0x100c748,%esi
mov    %rax,%rdi
callq  100b320 <_ZN12OutputStreamlsEPKc>
mov    $0x100b6b0,%esi
mov    %rax,%rdi
callq  100b690 <_ZN12OutputStreamlsEPFRS_So_E>
callq  1009330 <_ZN6IOAPIC4initEv>
mov    $0x1024020,%edi
callq  10071c0 <_ZN8Keyboard6pluginEv>
mov    $0x10240a0,%edi
callq  1007a60 <_ZN5Mouse6pluginEv>
callq  1008350 <_ZN4Core5countEv>
callq  1003efo <_ZN20ApplicationProcessor4bootEv>
sti
nop
callq  1008330 <_ZN4Core5getIDEv>
mov    %eax,%edi
add    $0x10245a8,%rdi
callq  100b780 <_ZN11Application6actionEv>
mov    $0x10245a0,%edi
callq  100b7do <_ZN19KeyboardApplication6actionEv>
xor    %eax,%eax
add    $0x8,%rsp
retq
```

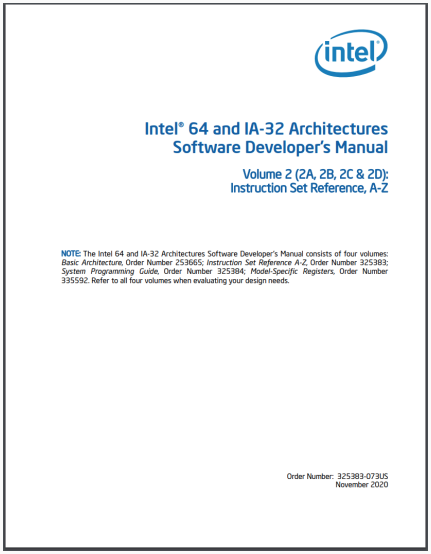
- Schnittstelle der **ISA**
- **Mnemonics** statt Bytes
- 1:1-Übersetzung der Befehle in Maschinencode durch Assembler

Befehle der x64-Architektur



- Dokumentation der Instruktionen:
  Intel-Manual, Volume 2

Befehle der x64-Architektur



intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

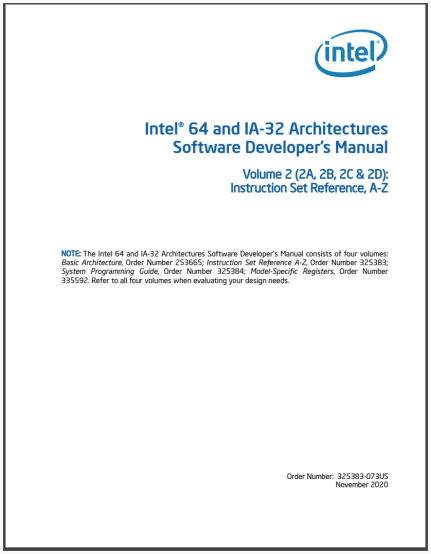
Volume 2 (2A, 2B, 2C & 2D):
Instruction Set Reference, A-Z

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325383-073US
November 2020

- Dokumentation der Instruktionen: Intel-Manual, Volume 2
- $>$ **2300** Seiten

Befehle der x64-Architektur



- Dokumentation der Instruktionen: Intel-Manual, Volume 2
- $>$ **2300** Seiten

Syntaxunterschiede bei x86-Assembler

Intel:

```
mov ebx, 0x17
```

(Ziel, Quelle)

`objdump -M intel`

AT&T:

```
mov $0x17, %ebx
```

(Quelle, Ziel)

Standard bei `objdump`

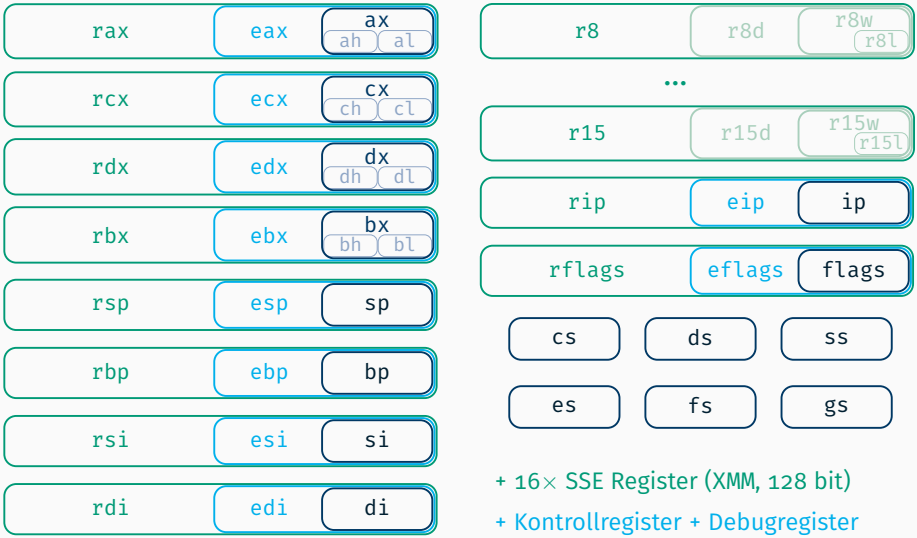
Eine einfache Einführung

- Ab hier: **Intel-Syntax** (`<Op> <Ziel> <Quelle>`)
  - Wird von NASM (Netwide Assembler) verwendet

Eine einfache Einführung

- Ab hier: **Intel-Syntax** (`<Op> <Ziel> <Quelle>`)
  - Wird von NASM (Netwide Assembler) verwendet
- Grundlegender Aufbau der x64-Architektur
  - 16 General-Purpose-Register
  - Instruktionszeiger
  - Stack Pointer (zeigt auf **zuletzt abgelegtes** Datum)

x64 (Long Mode) Register



+ 16× SSE Register (XMM, 128 bit)

+ Kontrollregister + Debugregister

Programmieren in Assembler: `mov`

Berechnungen generell auf Registern

Wie kommen Daten dort hin?

Programmieren in Assembler: `mov`

Berechnungen generell auf Registern

Wie kommen Daten dort hin?

➔ `mov`-Instruktion

Register ← **Konstante** `mov rax, 42`

Programmieren in Assembler: `mov`

Berechnungen generell auf Registern

Wie kommen Daten dort hin?

➔ `mov`-Instruktion

Register ← Konstante	`mov rax, 42`
Register ← Register	`mov rax, rcx`

Programmieren in Assembler: mov

Berechnungen generell auf Registern

Wie kommen Daten dort hin?

→ mov-Instruktion

Register ← Konstante	`mov rax, 42`
Register ← Register	`mov rax, rcx`
Register ↔ Speicher	`mov rax, [0xb8000]`
	`mov [0xb8000 + 4], rax`
	`mov rax, [rcx + rdx]`
	`mov rax, [rsp]`
	~~`mov [0xb8000], [0xb8002]`~~

Besonderheiten bei `mov`

- Was passiert beim Schreiben von Teilen von Registern?

Besonderheiten bei `mov`

- Was passiert beim Schreiben von Teilen von Registern?

  `mov eax, XX` `eax` wird gesetzt, obere Hälfte **genullt**

Besonderheiten bei `mov`

- Was passiert beim Schreiben von Teilen von Registern?

  `mov eax, XX` `eax` wird gesetzt, obere Hälfte **genullt**

  `mov ax, XX` `ax` wird gesetzt, **Rest unverändert**

  `mov al, XX` `al` wird gesetzt, **Rest unverändert**

  Falls Rest gesetzt werden muss: `movzx`, `movsx`

Details zur Performance bei partiellem Schreiben: → StackOverflow

movfuscator - the single instruction C compiler

mov auf x86 ist Turing-vollständig!

movfuscator - the single instruction C compiler

mov auf x86 ist Turing-vollständig!

```
int is_prime(int x)
{
  int  i;
  if (x==1) {
    return 0;
  }
  if  (x==2)  {
    return 1;
  }
  for  (i=2; i*i<=x; i++) {
    if  (x%i==0)  {
      return  0;
    }
  }
  return 1;
}
```

movfuscator - the single instruction C compiler

mov auf x86 ist Turing-vollständig!

```c
int is_prime(int x)
{
  int  i;
  if (x==1) {
    return 0;
  }
  if  (x==2)  {
    return 1;
  }
  for  (i=2; i*i<=x; i++) {
    if  (x%i==0)  {
      return  0;
    }
  }
  return 1;
}
```

```
<is_prime>:
  push  ebp
  mov   ebp,esp
  sub   esp,0x10
  cmp   DWORD PTR [ebp+0x8],0x1
  jne   8048490 <is_prime+0x13>
  mov   eax,0x0
  jmp   80484cf <is_prime+0x52>
  cmp   DWORD PTR [ebp+0x8],0x2
  jne   804849d <is_prime+0x20>
  mov   eax,0x1
  jmp   80484cf <is_prime+0x52>
  mov   DWORD PTR [ebp-0x4],0x2
  jmp   80484be <is_prime+0x41>
  mov   eax,DWORD PTR [ebp+0x8]
  cdq
  idiv  DWORD PTR [ebp-0x4]
  mov   eax,edx
  test  eax,eax
  jne   80484ba <is_prime+0x3d>
  mov   eax,0x0
  jmp   80484cf <is_prime+0x52>
  add   DWORD PTR [ebp-0x4],0x1
  mov   eax,DWORD PTR [ebp-0x4]
  imul  eax,DWORD PTR [ebp-0x4]
  cmp   eax,DWORD PTR [ebp+0x8]
  jle   80484a6 <is_prime+0x29>
  mov   eax,0x1
  leave
  ret
```

movfuscator - the single instruction C compiler

mov auf x86 ist Turing-vollständig!

```c
int is_prime(int x)
{
  int  i;
  if (x==1) {
    return 0;
  }
  if  (x==2)  {
    return 1;
  }
  for  (i=2; i*i<=x; i++) {
    if  (x%i==0)  {
      return  0;
    }
  }
  return 1;
}
```

```
<is_prime>:
  push  ebp
  mov   ebp,esp
  sub   esp,0x10
  cmp   DWORD PTR [ebp+0x8],0x1
  jne   8048490 <is_prime+0x13>
  mov   eax,0x0
  jmp   80484cf <is_prime+0x52>
  cmp   DWORD PTR [ebp+0x8],0x2
  jne   804849d <is_prime+0x20>
  mov   eax,0x1
  jmp   80484cf <is_prime+0x52>
  mov   DWORD PTR [ebp-0x4],0x2
  jmp   80484be <is_prime+0x41>
  mov   eax,DWORD PTR [ebp+0x8]
  cdq
  idiv  DWORD PTR [ebp-0x4]
  mov   eax,edx
  test  eax,eax
  jne   80484ba <is_prime+0x3d>
  mov   eax,0x0
  jmp   80484cf <is_prime+0x52>
  add   DWORD PTR [ebp-0x4],0x1
  mov   eax,DWORD PTR [ebp-0x4]
  imul  eax,DWORD PTR [ebp-0x4]
  cmp   eax,DWORD PTR [ebp+0x8]
  jle   80484a6 <is_prime+0x29>
  mov   eax,0x1
  leave
  ret
```

```
<is_prime>:
  mov  eax,ds:0x83fc638
  mov  edx,0x88048744
  mov  ds:0x81fc4c0,eax
  mov  DWORD PTR ds:0x81fc4c4,edx
  mov  eax,0x0
  mov  ecx,0x0
  mov  edx,0x0
  mov  al,ds:0x81fc4c0
  mov  ecx,DWORD PTR [eax*4+0x8056ad0]
  mov  dl,BYTE PTR ds:0x81fc4c4
  mov  dl,BYTE PTR [ecx+edx*1]
  mov  DWORD PTR ds:0x81fc4b0,edx
  mov  al,ds:0x81fc4c1
  mov  ecx,DWORD PTR [eax*4+0x8056ad0]
  mov  dl,BYTE PTR ds:0x81fc4c5
  mov  dl,BYTE PTR [ecx+edx*1]
  mov  DWORD PTR ds:0x81fc4b4,edx
  mov  al,ds:0x81fc4c2
  mov  ecx,DWORD PTR [eax*4+0x8056ad0]
  mov  dl,BYTE PTR ds:0x81fc4c6
  mov  dl,BYTE PTR [ecx+edx*1]
  mov  DWORD PTR ds:0x81fc4b8,edx
  mov  al,ds:0x81fc4c3
  mov  ecx,DWORD PTR [eax*4+0x8056ad0]
  mov  dl,BYTE PTR ds:0x81fc4c7
  mov  dl,BYTE PTR [ecx+edx*1]
  mov  DWORD PTR ds:0x81fc4bc,edx
  mov  eax,ds:0x81fc4b0
  mov  edx,DWORD PTR ds:0x81fc4b4
<...> (5710 more lines)
```

Operationen mit dem Stack

Register ← Stack

```
pop rax
```

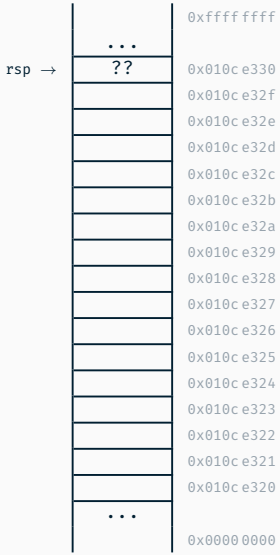
```
mov rax, [rsp]
add rsp, 8
```

Stack ← Register

```
push rax
```

```
sub rsp, 8
mov [rsp], rax
```

Stack-Operationen am Beispiel

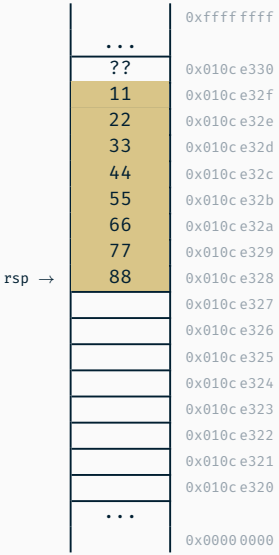


0x1122334455667788 rax

0x???????????????? rcx

```
long long i = 0x112233445566778ll;
```

Stack-Operationen am Beispiel

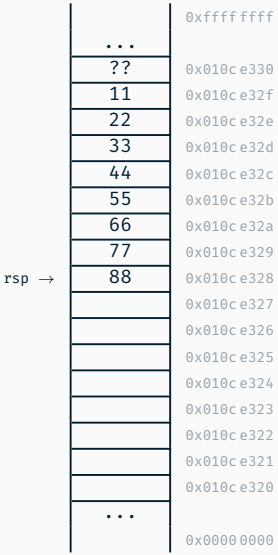


0x1122334455667788 rax

0x???????????????? rcx

```
    mov rax, 0x1122334455667788
→   push rax
    mov eax, 0xaabbccdd
    push rax
    pop rcx
    mov rcx, 0
    push rcx
```

Stack-Operationen am Beispiel

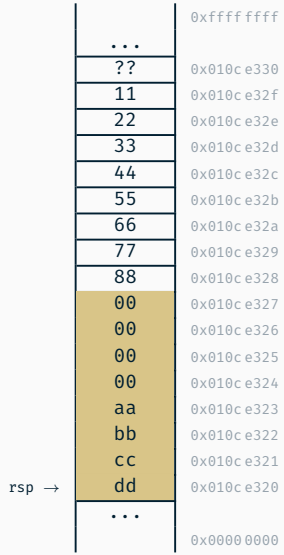


rax

rcx

```
    mov rax, 0x1122334455667788
    push rax
→   mov eax, 0xaabbccdd
    push rax
    pop rcx
    mov rcx, 0
    push rcx
```

Stack-Operationen am Beispiel

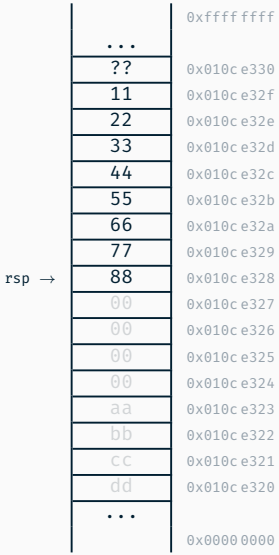


0x00000000aabbccdd rax

0x???????????????? rcx

```
    mov rax, 0x1122334455667788
    push rax
    mov eax, 0xaabbccdd
→   push rax
    pop rcx
    mov rcx, 0
    push rcx
```

Stack-Operationen am Beispiel

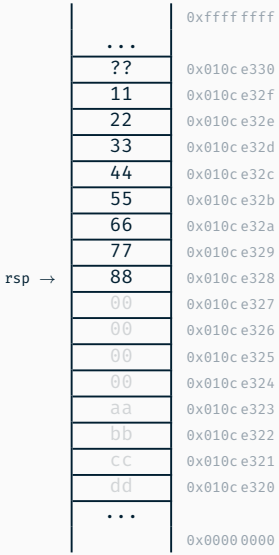


0x00000000aabbccdd rax

0x00000000aabbccdd rcx

```
    mov rax, 0x1122334455667788
    push rax
    mov eax, 0xaabbccdd
    push rax
→   pop rcx
    mov rcx, 0
    push rcx
```

Stack-Operationen am Beispiel

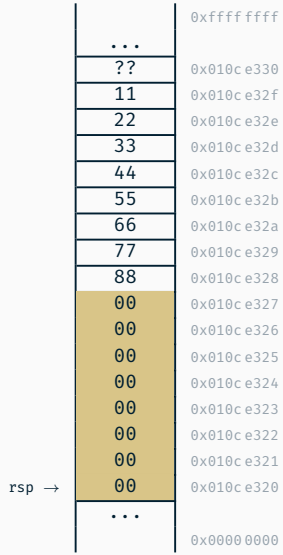


0x00000000aabbccdd rax

0x0000000000000000 rcx

```
mov rax, 0x1122334455667788
push rax
mov eax, 0xaabbccdd
push rax
pop rcx
→ mov rcx, 0
push rcx
```

Stack-Operationen am Beispiel



0x00000000aabbccdd rax

0x0000000000000000 rcx

```
    mov rax, 0x1122334455667788
    push rax
    mov eax, 0xaabbccdd
    push rax
    pop rcx
    mov rcx, 0
→   push rcx
```

Mathematische Operationen

Addition/Subtraktion

```
add rax, 4              x += 4;
sub qword [rax], rcx    *x -= rcx;
inc rax                 x++;
dec rax                 x--;
```

Mathematische Operationen

Addition/Subtraktion	`add rax, 4`	`x += 4;`
	`sub qword [rax], rcx`	`*x -= rcx;`
	`inc rax`	`x++;`
	`dec rax`	`x--;`
Bit-Operationen	`and rax, 0xff`	`x &= 0xff;`
	`or rax, [rcx]`	`x |= *rcx;`
	`xor rax, 0xaa`	`x ^= 0xaa;`
	`not rax`	`x = !x;`
	`neg rcx`	`x = ~x;`
	`shl rax, 1`	`x <<= 1;`
	`shr rax, 1`	`x >>= 1;`

Kontrollfluss-Beeinflussung

Labels als Sprungziele

```
block:
  add rcx, 1
  <...>
```

Einfache Sprünge

```
jmp block
```

Bedingte Sprünge

```
je/jne/jg/jge/jl/jle/jz block
```

Kontrollfluss-Beeinflussung

Labels als Sprungziele

```
block:
  add rcx, 1
  <...>
```

Einfache Sprünge

```
jmp block
```

Bedingte Sprünge

```
je/jne/jg/jge/jl/jle/jz block
```

Woher kommt die Bedingung für einen bedingten Sprung?

Kontrollfluss-Beeinflussung

Labels als Sprungziele

```
block:
  add rcx, 1
  <...>
```

Einfache Sprünge `jmp block`

Bedingte Sprünge `je/jne/jg/jge/jl/jle/jz block`

Woher kommt die Bedingung für einen bedingten Sprung?

- Statusregister (`rflags`)
- Mathematische Operationen und Vergleiche setzen Flags im Statuswort

Vergleiche

```
cmp rax, rdx
cmp rax, 0x10
≅ sub rax, 0x10
```

(aber rax unverändert)

Funktionsaufrufe

Funktionsaufruf

```
call function
```

```
push <next RIP>
jmp function
```

Rückkehr aus Funktion

```
ret
```

```
pop rip
```

Rückkehr aus Interrupt

```
iretq
```

Stellt Hardware-Kontext (`rip, cs, rflags`) wieder her, siehe Übung 2

Application Binary Interface (ABI)

Verschiedene Konventionen für

- Übergabe von Parametern
- Flüchtigkeit von Registern
- Umgang mit Stack in Aufrufer und aufgerufener Funktion

→ **x86** (32 Bit)

- 12 verschiedene Konventionen ☹
- Linux/C/GCC-Standard (`cdecl`):
  - Parameter auf dem Stack, letzte Parameter zuerst
  - Rückgabewert in `eax`
  - Aufrufer muss `eax`, `ecx`, `edx` sichern, falls Wert noch benötigt (**flüchtig**)

Application Binary Interface (ABI, x64)

- **x64**: nur noch zwei Konventionen (Microsoft und System V)
- System V Application Binary Interface:
  - 6 Parameter in Registern (`rdi, rsi, rdx, rcx, r8, r9`), Rest auf dem Stack
  - Rückgabewert in `rax` oder `rdx:rax`
  - **Nicht-flüchtige** Register: `rbx, rbp, r12 - r15`
  - C++-Code: **`this`** ist impliziter erster Parameter

Demotime!

Wir schreiben Code in Assembler