

Übung zu Betriebssysteme

x64-Assembler im Detail

6. Dezember 2022

Phillip Raffeck, Andreas Ziegler

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



Lehrstuhl für Verteilte Systeme
und Betriebssysteme

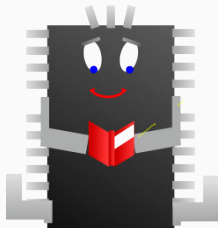


FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

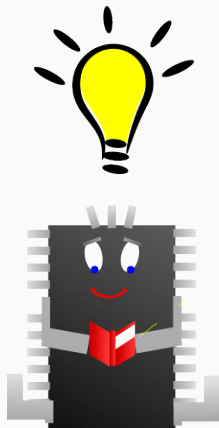
Mensch vs. CPU

```
48 83 ec 08
be 60 51 02 01
bf 60 55 02 01
e8 9d c2 ff ff
31 d2
be 23 00 00 00
bf c4 55 02 01
e8 6c f8 ff ff
48 8b 35 d5 33 00
bf 60 55 02 01
e8 2b 02 00 00
be 2d d9 00 01
48 89 c7
e8 1e 02 00 00
be 80 bb 00 01
48 89 c7
e8 11 05 00 00
e8 1c d7 ff ff
bf 60 50 02 01
e8 e2 ba ff ff
bf e0 50 02 01
e8 78 c5 ff ff
e8 63 da ff ff
e8 ce 89 ff ff
fb
90
[...]
```



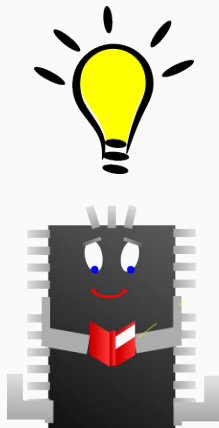
Mensch vs. CPU

```
48 83 ec 08
be 60 51 02 01
bf 60 55 02 01
e8 9d c2 ff ff
31 d2
be 23 00 00 00
bf c4 55 02 01
e8 6c f8 ff ff
48 8b 35 d5 33 00
bf 60 55 02 01
e8 2b 02 00 00
be 2d d9 00 01
48 89 c7
e8 1e 02 00 00
be 80 bb 00 01
48 89 c7
e8 11 05 00 00
e8 1c d7 ff ff
bf 60 50 02 01
e8 e2 ba ff ff
bf e0 50 02 01
e8 78 c5 ff ff
e8 63 da ff ff
e8 ce 89 ff ff
fb
90
[...]
```



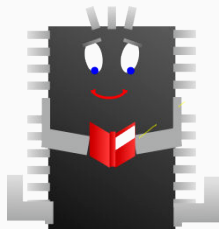
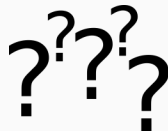
Mensch vs. CPU

```
48 83 ec 08
be 60 51 02 01
bf 60 55 02 01
e8 9d c2 ff ff
31 d2
be 23 00 00 00
bf c4 55 02 01
e8 6c f8 ff ff
48 8b 35 d5 33 00
bf 60 55 02 01
e8 2b 02 00 00
be 2d d9 00 01
48 89 c7
e8 1e 02 00 00
be 80 bb 00 01
48 89 c7
e8 11 05 00 00
e8 1c d7 ff ff
bf 60 50 02 01
e8 e2 ba ff ff
bf e0 50 02 01
e8 78 c5 ff ff
e8 63 da ff ff
e8 ce 89 ff ff
fb
90
[...]
```



Mensch vs. CPU

```
48 83 ec 08
be 60 51 02 01
bf 60 55 02 01
e8 9d c2 ff ff
31 d2
be 23 00 00 00
bf c4 55 02 01
e8 6c f8 ff ff
48 8b 35 d5 33 00
bf 60 55 02 01
e8 2b 02 00 00
be 2d d9 00 01
48 89 c7
e8 1e 02 00 00
be 80 bb 00 01
48 89 c7
e8 11 05 00 00
e8 1c d7 ff ff
bf 60 50 02 01
e8 e2 ba ff ff
bf e0 50 02 01
e8 78 c5 ff ff
e8 63 da ff ff
e8 ce 89 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;

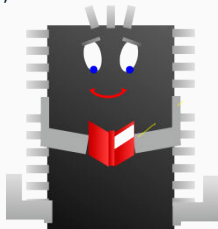
    ApplicationProcessor::boot();

    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;

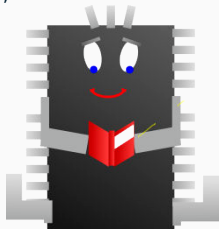
    ApplicationProcessor::boot();

    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;

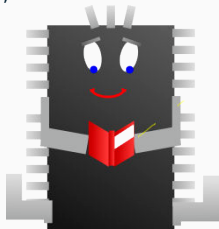
    ApplicationProcessor::boot();

    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;

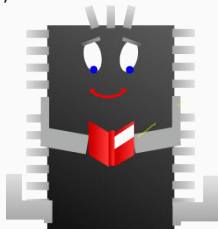
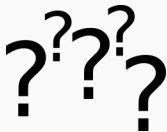
    ApplicationProcessor::boot();

    Core::Interrupt::enable();

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

    kapp.action();

    return 0;
}
```



Kompromiss: Assembler Language

```
sub    rsp,0x8
mov    esi,0x1025160
mov    edi,0x1025560
call   10078b0 <_ZN10TextStream7arrangeERS_PS_>
xor    edx,edx
mov    esi,0x23
mov    edi,0x10255c4
call   100ae90 <_ZN10TextWindow6setPosEii>
mov    rsi,QWORD PTR [rip+0x33d5] # 100ea00 <os_name>
mov    edi,0x1025560
call   100b860 <_ZN12OutputStreamlsEPKc>
mov    esi,0x100d92d
mov    rdi,rax
call   100b860 <_ZN12OutputStreamlsEPKc>
mov    esi,0x100bb80
mov    rdi,rax
call   100bb60 <_ZN12OutputStreamlsEPFRS_S0_E>
call   1008d70 <_ZN6IOAPIC4initEv>
mov    edi,0x1025060
call   1007140 <_ZN8Keyboard6pluginEv>
mov    edi,0x10250e0
call   1007be0 <_ZN5Mouse6pluginEv>
call   10090d0 <_ZN4Core5countEv>
call   1004040 <_ZN20ApplicationProcessor4bootEv>
sti
nop
[...]
```

- Schnittstelle der ISA

- Mnemonics statt Bytes

- 1:1-Übersetzung der Befehle in Maschinencode durch Assembler

Kompromiss: Assembler Language

```
sub    rsp,0x8                48 83 ec 08
mov    esi,0x1025160          be 60 51 02 01
mov    edi,0x1025560          bf 60 55 02 01
call   10078b0 <_ZN10TextStream7arrangeERS_PS_> e8 9d c2 ff ff
xor    edx,edx                31 d2
mov    esi,0x23                be 23 00 00 00
mov    edi,0x10255c4          bf c4 55 02 01
call   100ae90 <_ZN10TextWindow6setPosEii>     e8 6c f8 ff ff
mov    rsi,QWORD PTR [rip+0x33d5] # 100ea00 <os_name> 48 8b 35 d5 33 00 00
mov    edi,0x1025560          bf 60 55 02 01
call   100b860 <_ZN12OutputStreamlsEPKc>       e8 2b 02 00 00
mov    esi,0x100d92d          be 2d d9 00 01
mov    rdi,rax                48 89 c7
call   100b860 <_ZN12OutputStreamlsEPKc>       e8 1e 02 00 00
mov    esi,0x100bb80          be 80 bb 00 01
mov    rdi,rax                48 89 c7
call   100bb60 <_ZN12OutputStreamlsEPFRS_S0_E> e8 11 05 00 00
call   1008d70 <_ZN6IOAPIC4initEv>             e8 1c d7 ff ff
mov    edi,0x1025060          bf 60 50 02 01
call   1007140 <_ZN8Keyboard6pluginEv>         e8 e2 ba ff ff
mov    edi,0x10250e0          bf e0 50 02 01
call   1007be0 <_ZN5Mouse6pluginEv>           e8 78 c5 ff ff
call   10090d0 <_ZN4Core5countEv>             e8 63 da ff ff
call   1004040 <_ZN20ApplicationProcessor4bootEv> e8 ce 89 ff ff
sti                                         fb
nop                                         90
[...]
```



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: 325383-071US
October 2019

- Dokumentation der Instruktionen:
Intel-Manual, Volume 2

Befehle der x64-Architektur



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: 325383-071US
October 2019

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

Befehle der x64-Architektur



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: 325383-071US
October 2019

- 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
```

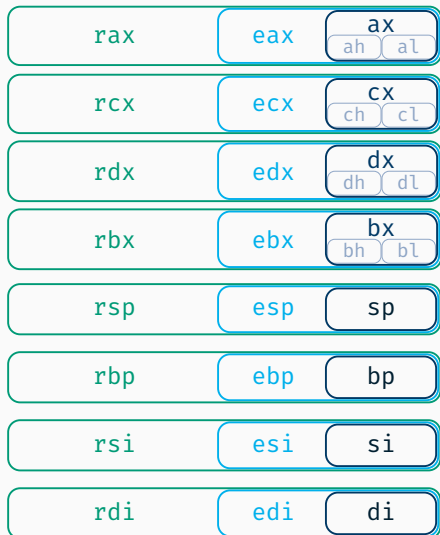
Eine einfache Einführung

- 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	<code>mov rax, 42</code>
Register ← Register	<code>mov rax, rcx</code>
Register ↔ Speicher	<code>mov rax, [0xb8000]</code>
	<code>mov [0xb8000 + 4], rax</code>
	<code>mov rax, [rcx + rdx]</code>
	<code>mov rax, [rsp]</code>
	<code>mov [0xb8000], [0xb8002]</code>

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!

→ <https://github.com/xoreaxeaxeax/movfuscator>

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;
}
```

→ <https://github.com/xoreaxeaxeax/movfuscator>

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;
}

<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
```

→ <https://github.com/xoreaxeaxeax/movfuscator>

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;
}
```

```
<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
        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
```

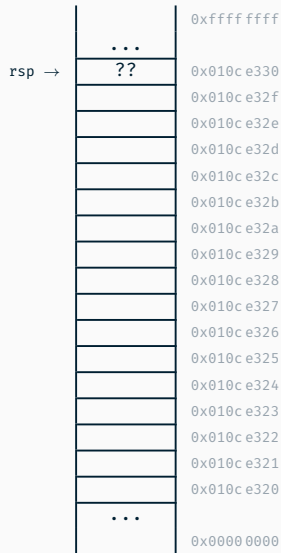
```
<is_prime>:
mov     eax,ds:0x83fc638
mov     edx,0x88048744
mov     ds:0x81fc4c0,eax
mov     DWORD PTR ds:0x81fc4c4,edx
mov     eax,0x0
mov     ecx,0x0
mov     edx,0x0
mov     al,ds:0x81fc4c0
mov     ecx,DWORD PTR [eax+4+0x8056ad0]
mov     dl,BYTE PTR ds:0x81fc4c4
mov     dl,BYTE PTR [ecx+edx+1]
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)
```

→ <https://github.com/xoreaxeaxeax/movfuscator>

Operationen mit dem Stack

Register ← Stack	<code>pop rax</code>
	<code>mov rax, [rsp]</code>
	<code>add rsp, 8</code>
Stack ← Register	<code>push rax</code>
	<code>sub rsp, 8</code>
	<code>mov [rsp], rax</code>

Stack-Operationen am Beispiel



0x1122334455667788

rax

0x????????????????

rcx

```
long long i = 0x1122334455667788ll;
```

Stack-Operationen am Beispiel

	...	0xffffffff
	??	0x010c e330
	11	0x010c e32f
	22	0x010c e32e
	33	0x010c e32d
	44	0x010c e32c
	55	0x010c e32b
	66	0x010c e32a
	77	0x010c e329
rsp →	88	0x010c e328
		0x010c e327
		0x010c e326
		0x010c e325
		0x010c e324
		0x010c e323
		0x010c e322
		0x010c e321
		0x010c e320
	...	0x0000 0000

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

	...	0xffffffff
	??	0x010c e330
	11	0x010c e32f
	22	0x010c e32e
	33	0x010c e32d
	44	0x010c e32c
	55	0x010c e32b
	66	0x010c e32a
	77	0x010c e329
rsp →	88	0x010c e328
		0x010c e327
		0x010c e326
		0x010c e325
		0x010c e324
		0x010c e323
		0x010c e322
		0x010c e321
		0x010c e320
	...	0x0000 0000

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

...	0xffffffff
??	0x010c e330
11	0x010c e32f
22	0x010c e32e
33	0x010c e32d
44	0x010c e32c
55	0x010c e32b
66	0x010c e32a
77	0x010c e329
88	0x010c e328
00	0x010c e327
00	0x010c e326
00	0x010c e325
00	0x010c e324
aa	0x010c e323
bb	0x010c e322
cc	0x010c e321
dd	0x010c e320
...	0x0000 0000

rsp →

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

	...	0xffffffff
	??	0x010c e330
	11	0x010c e32f
	22	0x010c e32e
	33	0x010c e32d
	44	0x010c e32c
	55	0x010c e32b
	66	0x010c e32a
	77	0x010c e329
rsp →	88	0x010c e328
	00	0x010c e327
	00	0x010c e326
	00	0x010c e325
	00	0x010c e324
	aa	0x010c e323
	bb	0x010c e322
	cc	0x010c e321
	dd	0x010c e320
	...	0x0000 0000

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

	...	0xffffffff
	??	0x010c e330
	11	0x010c e32f
	22	0x010c e32e
	33	0x010c e32d
	44	0x010c e32c
	55	0x010c e32b
	66	0x010c e32a
rsp →	77	0x010c e329
	88	0x010c e328
	00	0x010c e327
	00	0x010c e326
	00	0x010c e325
	00	0x010c e324
	aa	0x010c e323
	bb	0x010c e322
	cc	0x010c e321
	dd	0x010c e320
	...	0x0000 0000

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

...	0xffffffff
??	0x010c e330
11	0x010c e32f
22	0x010c e32e
33	0x010c e32d
44	0x010c e32c
55	0x010c e32b
66	0x010c e32a
77	0x010c e329
88	0x010c e328
00	0x010c e327
00	0x010c e326
00	0x010c e325
00	0x010c e324
00	0x010c e323
00	0x010c e322
00	0x010c e321
00	0x010c e320
...	0x0000 0000

rsp →

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	<code>add rax, 4</code>	<code>x += 4;</code>
	<code>sub qword [rax], rcx</code>	<code>*x -= rcx;</code>
	<code>inc rax</code>	<code>x++;</code>
	<code>dec rax</code>	<code>x--;</code>

Mathematische Operationen

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

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`
 \cong `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 wieder her
(`rip`, `cs`, `rflags`) → 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

Ein Wort zu inline-Assembly

```
inline void enable() {  
    asm volatile("sti\n\t nop\n\t" : : : "memory");  
}
```

Ein Wort zu inline-Assembly

```
inline void enable() {  
    asm volatile("sti\n\t nop\n\t" : : : "memory");  
}
```

**C++ N4917 „Working Draft, Standard for Programming Language C++“
Abschnitt 9.10**

The asm declaration is conditionally-supported; its meaning is implementation-defined.

Ein Wort zu inline-Assembly

```
inline void enable() {  
    asm volatile("sti\n\t nop\n\t" : : : "memory");  
}
```

**C++ N4917 „Working Draft, Standard for Programming Language C++“
Abschnitt 9.10**

The asm declaration is conditionally-supported; its meaning is implementation-defined.

fehleranfälliger, bspw. durch Compilerinfluss

Demotime!

Wir schreiben Code in Assembler