



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

MIN-Fakultät
Fachbereich Informatik



Rechnerstrukturen und Betriebssysteme

64-040 Vorlesung RSB
Kapitel 13: Assemblerebene

Norman Hendrich



Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Technische Aspekte Multimodaler Systeme

Wintersemester 2025/2026

Assembler-Programmierung

- Motivation

- Grundlagen der Assemblerebene

- x86 Assembler

 - Elementare Befehle + Adressierung

 - Operationen

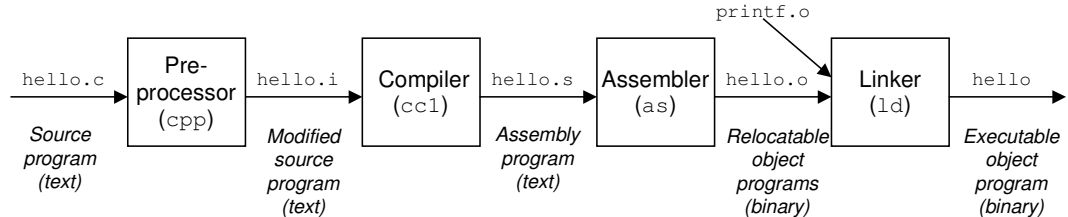
 - Kontrollfluss

 - Sprungbefehle und Schleifen

 - Funktionsaufrufe und Stack

- Linker und Loader

- Literatur



[BO15]

- ▶ verschiedene Repräsentationen des Programms
 - ▶ Hochsprache
 - ▶ Assembler
 - ▶ Maschinensprache
- ▶ Ausführung der Maschinensprache
 - ▶ von-Neumann Zyklus: Befehl holen, decodieren, ausführen
 - ▶ reale oder virtuelle Maschine

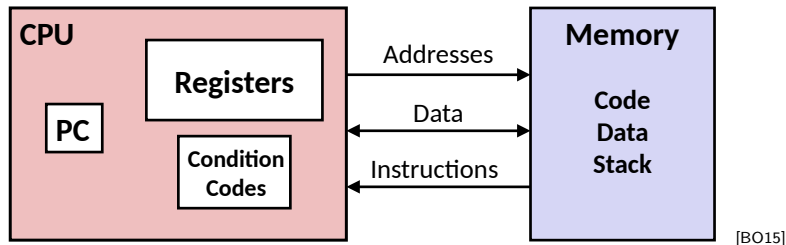
Programme werden nur noch selten in Assembler geschrieben

- ▶ Programmentwicklung in Hochsprachen weit produktiver
- ▶ Compiler/Tools oft besser als handcodierter Assembler

aber **Grundwissen** bleibt trotzdem **unverzichtbar**

- ▶ Verständnis des Ausführungsmodells auf der Maschinenebene
- ▶ Programmverhalten bei Fehlern / Debugging
 - ▶ das High-Level Sprachmodell ist dort nicht anwendbar
- ▶ Programmleistung verstärken
 - ▶ Ursachen für Programm-Ineffizienz verstehen
 - ▶ effiziente „maschinengerechte“ Datenstrukturen / Algorithmen
- ▶ Systemsoftware implementieren
 - ▶ Compilerbau: Maschinencode als Ziel
 - ▶ Betriebssysteme implementieren (Prozesszustände verwalten)
 - ▶ Gerätetreiber schreiben

- ▶ Beschränkung auf wesentliche Konzepte
 - ▶ GNU Assembler für x86-64 (Linux, 64-bit)
 - ▶ nur ein Datentyp: 64-bit Integer (long)
 - ▶ nur kleiner Subset des gesamten Befehlssatzes
- ▶ diverse nicht behandelte Themen
 - ▶ Speicherverwaltung der Datentypen
 - ▶ Behandlung von Makros
 - ▶ Implementierung eines Assemblers (2-pass)
 - ▶ Tipps für effizientes Programmieren
 - ▶ Befehle für die Systemprogrammierung (supervisor mode)
 - ▶ x86 Gleitkommabefehle, Befehlssatzerweiterungen (MMX, SSE, AVX etc.)
 - ▶ ...



beobachtbare Zustände

- ▶ Programmzähler (*Instruction Pointer*)
 - ▶ Adresse der nächsten Anweisung
- ▶ Registerbank
 - ▶ häufig benutzte Programmdaten
- ▶ Zustandscodes
 - ▶ Statusinformationen über die letzte ALU Operation
 - ▶ für bedingte Sprünge benötigt (*Conditional Branch*)

x86-64 rip Register

rax...rbp Register

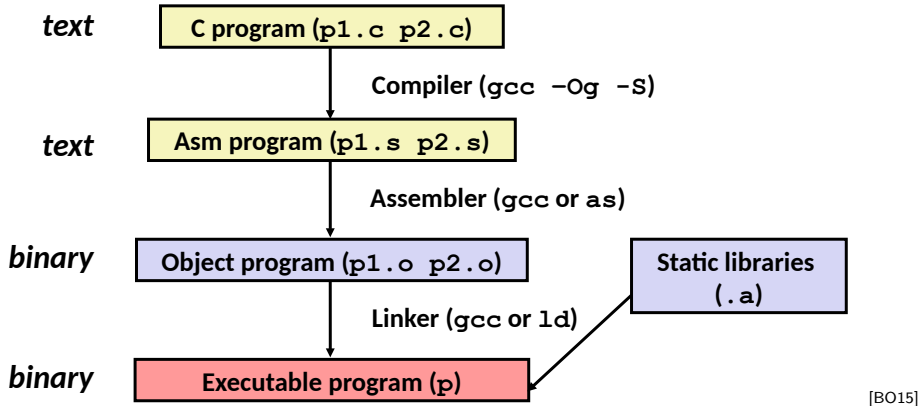
r8...r15 Register

EFLAGS Register

► Speicher

- byteweise adressierbares Array
- pro Task: Code, Nutzerdaten, (einige) OS Daten
- —"— Kellerspeicher für Unterprogrammaufrufe „Stack“
- —"— Adressraum dynamisch allozierter Datenstrukturen „Heap“

Umwandlung von C in Objektcode



[BO15]

Compilieren zu Assemblercode: Funktion sum()

sum.c

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

sum.s

```
sumstore:  
    pushq    %rbx  
    movq     %rdx, %rbx  
    call     plus  
    movq     %rax, (%rbx)  
    popq     %rbx  
    ret
```

[BO15]

- ▶ Befehl `gcc -Og -S sum.c`
- ▶ Erzeugt `sum.s`

Assemblercode: sum.s

```
.globl sumstore
.type sumstore, @function
sumstore:
.LFB35:
.cfi_startproc
pushq %rbx
.cfi_def_cfa_offset 16
.cfi_offset 3, -16
movq %rdx, %rbx
call plus
movq %rax, (%rbx)
popq %rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE35:
.size sumstore, .-sumstore
```

```
.globl sumstore
.type sumstore, @function
sumstore:
.LFB35:
.cfi_startproc
pushq %rbx
.cfi_def_cfa_offset 16
.cfi_offset 3, -16
movq %rdx, %rbx
call plus
movq %rax, (%rbx)
popq %rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE35:
.size sumstore, .-sumstore
```

```
sumstore:
    pushq    %rbx
    movq     %rdx, %rbx
    call     plus
    movq     %rax, (%rbx)
    popq     %rbx
    ret
```

- alles was mit „.“ beginnt: Label, Anweisungen für Linker

- ▶ hardwarenahe Programmierung: Zugriff auf kompletten Befehlssatz und alle Register einer Maschine
- ▶ je ein Befehl pro Zeile
 - ▶ **Mnemonics** für die einzelnen Maschinenbefehle
 - ▶ Konstanten als Dezimalwerte oder Hex-Werte
 - ▶ eingängige Namen für alle Register
 - ▶ Adressen für alle verfügbaren Adressierungsarten
 - ▶ Konvention bei gcc/as x86: Ziel einer Operation steht rechts
- ▶ symbolische **Label** für Sprungadressen
 - ▶ Verwendung in Sprungbefehlen
 - ▶ globale Label definieren Einsprungpunkte für den Linker/Loader

- ▶ nur die von der Maschine unterstützten „primitiven“ Daten
 - ▶ keine Aggregattypen wie Arrays, Strukturen oder Objekte
 - ▶ nur fortlaufend adressierbare Bytes im Speicher
 - ▶ Ganzzahl-Daten, z.B. 1, 2, 4 oder 8 Bytes
 - ▶ Datenwerte für Variablen
 - ▶ positiv oder vorzeichenbehaftet
 - ▶ Textzeichen (ASCII, Unicode)
 - ▶ Gleitkomma-Daten mit 4 oder 8 Bytes
 - ▶ Adressen bzw. „Pointer“
- | |
|-------------------------------|
| 8 ... 64 bits |
| int/long/long long |
| signed/unsigned |
| char |
| float/double |
| untypisierte Speicheradressen |

- ▶ arithmetische/logische Funktionen auf Registern und Speicher
 - ▶ Addition/Subtraktion, Multiplikation usw.
 - ▶ bitweise Logische- und Schiebe-Operationen
- ▶ Datentransfer zwischen Speicher und Registern
 - ▶ Daten aus Speicher in Register laden
 - ▶ Registerdaten im Speicher ablegen
 - ▶ ggf. auch Zugriff auf Spezial-/OS-register
- ▶ Kontrolltransfer
 - ▶ unbedingte / bedingte Sprünge
 - ▶ Unterprogrammaufrufe: Sprünge zu/von Prozeduren
 - ▶ Interrupts, Exceptions, System-Calls
- ▶ Makros: Folge von Assemblerbefehlen

Objektcode: Funktion `sumstore()`

- ▶ 14 Bytes Programmcode
- ▶ x86-Instruktionen mit 1-, 3- oder 5 Bytes
Erklärung s.u.
- ▶ Startadresse: `0x400595`
- ▶ vom Compiler/Assembler gewählt

`0x0400595:`

`0x53`

`0x48`

`0x89`

`0xd3`

`0xe8`

`0xf2`

`0xff`

`0xff`

`0xff`

`0x48`

`0x89`

`0x03`

`0x5b`

`0xc3`

Assembler

- ▶ übersetzt `.s` zu `.o`
- ▶ binäre Codierung jeder Anweisung
- ▶ (fast) vollständiges Bild des ausführbaren Codes
- ▶ keine Verknüpfungen zu Code aus anderen Dateien / zu Bibliotheksfunktionen

Linker / Binder

- ▶ löst Referenzen zwischen Dateien auf
- ▶ kombiniert mit statischen Laufzeit-Bibliotheken
 - ▶ z.B. Code für Bibliotheksfunktionen: `malloc`, `printf` etc.
- ▶ manche Bibliotheken sind *dynamisch* verknüpft
 - ▶ Verknüpfung wird beim Laden in den Speicher, bzw. zur Laufzeit erstellt
 - ▶ gemeinsame Nutzung von Standardbibliotheken

Beispiel: Maschinenbefehl für Speichern

- ▶ C-Code

```
*dest = t;
```

- ▶ speichert Wert t nach Adresse aus dest

- ▶ Assembler

```
movq %rax, (%rbx)
```

- ▶ Kopiere einen 8-Byte Wert in den Hauptspeicher
 - ▶ *Quad*-Worte in x86-64 Terminologie

- ▶ Operanden

t:	Register	%rax
dest:	Register	%rbx
*dest:	Speicher	M[%rbx]

- ▶ Objektcode (x86-Befehlssatz)

```
0x40059e: 48 89 03
```

- ▶ 3-Byte Befehl
 - ▶ an Speicheradresse 0x40059e

[BO15]

```
0000000000400595 <sumstore>:  
400595: 53                push    %rbx  
400596: 48 89 d3          mov     %rdx,%rbx  
400599: e8 f2 ff ff ff    callq   400590 <plus>  
40059e: 48 89 03          mov     %rax, (%rbx)  
4005a1: 5b                pop     %rbx  
4005a2: c3                retq
```

- ▶ `objdump -d ...`
 - ▶ Werkzeug zur Untersuchung des Objektcodes
 - ▶ rekonstruiert aus Binärcode den Assemblercode
 - ▶ kann auf vollständigem, ausführbarem Programm *<executable>* oder einer `.o` Datei ausgeführt werden

Was kann „disassembliert“ werden?

[BO15]

```
% objdump -d WINWORD.EXE
```

```
WINWORD.EXE:      file format pei-i386
```

```
No symbols in "WINWORD.EXE".
```

```
Disassembly of section .text:
```

```
30001000 <.text>:
```

```
30001000:
```

```
30001001:
```

```
30001003:
```

```
30001005:
```

```
3000100a:
```

**Reverse engineering forbidden by
Microsoft End User License Agreement**

- ▶ alles, was als ausführbarer Code interpretiert werden kann
- ▶ Disassembler untersucht Bytes und rekonstruiert Assemblerquelle (soweit möglich)

- ▶ Adressierungsarten
- ▶ arithmetische Operationen
- ▶ Statusregister
- ▶ Umsetzung von Programmstrukturen

Einschränkungen

- ▶ Beispiele nutzen nur die 64-bit Datentypen
long bei Linux (unter Windows nur 4-Byte!)
 - ▶ x86-64 wird wie 16-Register 64-bit Maschine benutzt (=RISC)
 - ▶ CISC Komplexität und Tricks bewusst vermieden
- ▶ Beispiele nutzen gcc/as Syntax (vs. Microsoft, Intel)

Grafiken und Beispiele dieses Abschnitts sind aus

R.E. Bryant, D.R. O'Hallaron: *Computer systems – A programmers perspective* [BO15]

bzw. dem zugehörigen Foliensatz

Datentransfer „move“

- ▶ Format: `movq <src>, <dst>`
- ▶ transferiert ein 8-Byte „long“ Wort
- ▶ sehr häufige Instruktion
- ▶ Typ der Operanden
 - ▶ Immediate: Konstante, ganzzahlig
 - ▶ wie C-Konstante, aber mit dem Präfix \$
 - ▶ z.B.: `$0x400`, `$-533`
 - ▶ codiert mit 1, 2 oder 4 Bytes
 - ▶ Register: 16 Ganzzahl-Register
 - ▶ `%rsp` (ggf. auch `%rbp`) für spezielle Aufgaben reserviert
 - ▶ z.T. Spezialregister für andere Anweisungen
 - ▶ Speicher: 8 konsekutive Speicherbytes
 - ▶ zahlreiche Adressmodi

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%r8`

...

`%r15`

movq Operanden-Kombinationen

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4,%rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

► Mem-Mem Kombination nicht möglich

movq: Operanden/Adressierungsarten

- ▶ Immediate: $\$x \rightarrow x$
 - ▶ positiver (oder negativer) Integerwert
- ▶ Register: $R \rightarrow \text{Reg}[R]$
 - ▶ Inhalt eines der 16 Universalregister `%rax...%r15`
Registername R beginnt immer mit %
- ▶ Normal: $(R) \rightarrow \text{Mem}[\text{Reg}[R]]$
 - ▶ Register R spezifiziert die Speicheradresse
 - ▶ Beispiel: `movq (%rcx), %rax`
- ▶ Displacement: $D(R) \rightarrow \text{Mem}[\text{Reg}[R]+D]$
 - ▶ Register R als Basis-Speicheradresse
 - ▶ Konstantes „Displacement“ D spezifiziert den „Offset“
 - ▶ Beispiel: `movq 8(%rbp), %rdx`

Beispiel: Funktion swap()

```
void swap
(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Register	Funktion
%rdi	Argument xp
%rsi	Argument yp
%rax	t0
%rdx	t1

Funktionsweise von swap()

Register

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

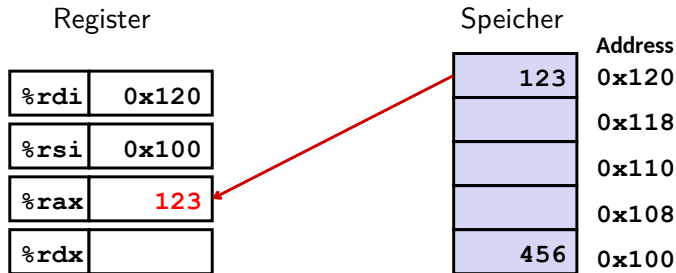
Speicher

Address
123
0x120
0x118
0x110
0x108
456
0x100

swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

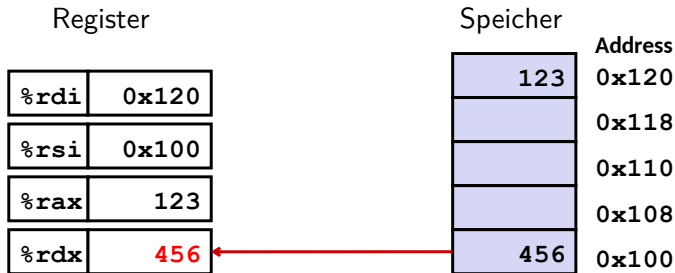
Funktionsweise von swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

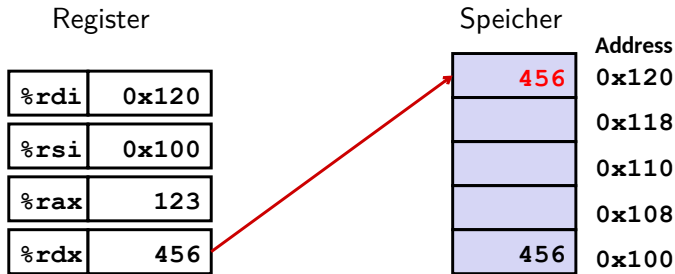
Funktionsweise von swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

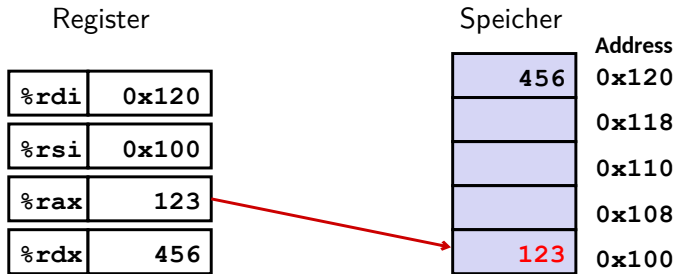
Funktionsweise von swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Funktionsweise von swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

► allgemeine Form

- $\text{Imm}(\text{Rb}, \text{Ri}, \text{S}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{S} * \text{Reg}[\text{Ri}] + \text{Imm}]$
 - $\langle \text{Imm} \rangle$ Offset
 - $\langle \text{Rb} \rangle$ Basisregister: eines der 16 Integer-Register
 - $\langle \text{Ri} \rangle$ Indexregister: jedes außer %rsp
%rbp grundsätzlich möglich, jedoch unwahrscheinlich
 - $\langle \text{S} \rangle$ Skalierungsfaktor 1, 2, 4 oder 8

► gebräuchlichste Fälle

- $(\text{Rb}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}]]$
- $\text{Imm}(\text{Rb}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{Imm}]$
- $(\text{Rb}, \text{Ri}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{Reg}[\text{Ri}]]$
- $\text{Imm}(\text{Rb}, \text{Ri}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{Reg}[\text{Ri}] + \text{Imm}]$
- $(\text{Rb}, \text{Ri}, \text{S}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{S} * \text{Reg}[\text{Ri}]]$

Beispiel: Adressberechnung

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8 (%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

► binäre Operatoren

Format	Berechnung
<code>addq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle + \langle src \rangle$
<code>subq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle - \langle src \rangle$
<code>imulq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle * \langle src \rangle$
<code>salq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \ll \langle src \rangle$ auch <code>shlq</code>
<code>sarq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \gg \langle src \rangle$ arithmetisch
<code>shrq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \gg \langle src \rangle$ logisch
<code>xorq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \wedge \langle src \rangle$
<code>andq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \& \langle src \rangle$
<code>orq</code> $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \mid \langle src \rangle$

► unäre Operatoren

Format	Berechnung
incq $\langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle + 1$
decq $\langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle - 1$
negq $\langle dst \rangle$	$\langle dst \rangle = -\langle dst \rangle$
notq $\langle dst \rangle$	$\langle dst \rangle = \sim \langle dst \rangle$

► leaq-Befehl: *load effective address*

leaq $\langle src \rangle, \langle dst \rangle$

- Adressberechnung für (späteren) Ladebefehl
- Speichert die Adressausdruck $\langle src \rangle$ in Register $\langle dst \rangle$
 $Imm(Rb, Ri, S) \rightarrow Reg[Rb] + S * Reg[Ri] + Imm$
- wird oft von Compilern für arithmetische Berechnung genutzt
s. Beispiele

Beispiel: arithmetische Operationen

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

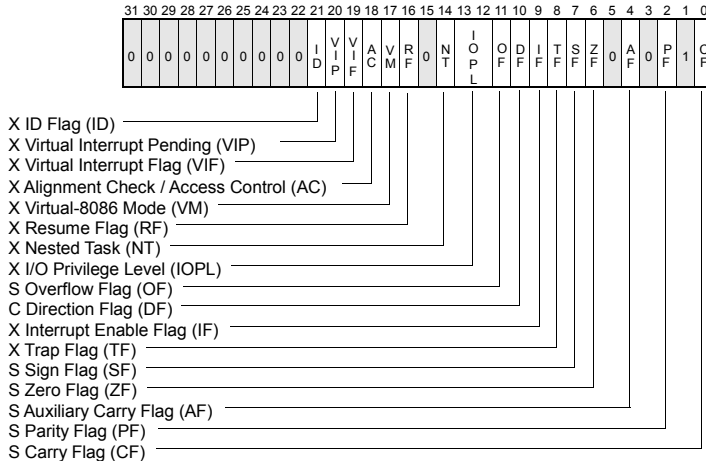
arith:

```
leaq    (%rdi,%rsi), %rax    # t1
addq    %rdx, %rax          # t2
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx            # t4
leaq    4(%rdi,%rdx), %rcx   # t5
imulq    %rcx, %rax          # rval
ret
```

Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z, t4
%rax	t1, t2, rval
%rcx	t5

- ▶ Zustandscodes
 - ▶ Setzen
 - ▶ Testen
- ▶ Ablaufsteuerung
 - ▶ Verzweigungen: „If-then-else“
 - ▶ Schleifen: „Loop“-Varianten
 - ▶ Mehrfachverzweigungen: „Switch“

x86: EFLAGS Register



S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

 Reserved bit positions. DO NOT USE.
Always set to values previously read.

► x86-64: RFLAGS $\hat{=}$ EFLAGS,
mit „0“ erweitert

[IA6]

Prozessor aus Sicht des Programmierers

- ▶ temporäre Daten, Standardregister
`%rax, ...`
- ▶ Top of Stack
`%rsp`
- ▶ Programmzähler
`%rip`
- ▶ Flag-Bits
CF, ZF, SF, OF

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip`

CF	ZF	SF	OF
----	----	----	----

- ▶ vier relevante „Flags“ im Statusregister EFLAGS/RFLAGS
 - ▶ CF Carry Flag
 - ▶ ZF Zero Flag
 - ▶ SF Sign Flag
 - ▶ OF Overflow Flag

1. implizite Aktualisierung durch arithmetische Operationen

- ▶ Beispiel: `addq <src>, <dst>` in C: `t=a+b`
 - ▶ CF höchstwertiges Bit generiert Übertrag: Unsigned-Überlauf
 - ▶ ZF wenn $t = 0$
 - ▶ SF wenn $t < 0$
 - ▶ OF wenn das Zweierkomplement überläuft
 $(a > 0 \ \&\& \ b > 0 \ \&\& \ t < 0) \ || \ (a < 0 \ \&\& \ b < 0 \ \&\& \ t \geq 0)$

2. explizites Setzen durch Vergleichsoperation

- ▶ Beispiel: `cmpq <src2>, <src1>`
wie Berechnung von $\langle src1 \rangle - \langle src2 \rangle$ (`subq <src2>, <src1>`)
jedoch ohne Abspeichern des Resultats
- ▶ CF höchstwertiges Bit generiert Übertrag
- ▶ ZF setzen wenn $src1 = src2$
- ▶ SF setzen wenn $(src1 - src2) < 0$
- ▶ OF setzen wenn das Zweierkomplement überläuft
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (a - b) < 0) \ || \ (a < 0 \ \&\& \ b > 0 \ \&\& \ (a - b) \geq 0)$

3. explizites Setzen durch Testanweisung

- ▶ Beispiel: `testq <src2>, <src1>`
wie Berechnung von $\langle src1 \rangle \& \langle src2 \rangle$
jedoch ohne Abspeichern des Resultats

`(andq <src2>, <src1>)`

⇒ hilfreich, wenn einer der Operanden eine Bitmaske ist

- ▶ ZF setzen wenn $src1 \& src2 = 0$
- ▶ SF setzen wenn $src1 \& src2 < 0$

Zustandscodes lesen: set...-Befehle

- ▶ Befehle setzen ein einzelnes Byte (LSB) in Universalregister
- ▶ die anderen 7-Bytes werden nicht verändert

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~ (SF^OF) & ~ZF	Greater (Signed)
setge	~ (SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF) ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Beispiel: Zustandscodes lesen

- ▶ ein-Byte Zieloperand (Register, Speicher)
- ▶ meist kombiniert mit `movzbl` (*move with zero-extend byte to long*)
also Löschen der Bits 31...8

```
int gt (long x, long y)
{
    return x > y;
}
```

```
cmpq    %rsi, %rdi    # Compare x:y
setg     %al           # Set when >
movzbl  %al, %eax      # Zero rest of %rax
ret
```

Sprünge („Jump“): j...-Befehle

- unbedingter- / bedingter Sprung (abhängig von Zustandscode)

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF) & ~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF) ZF	Less or Equal (Signed)
ja	~CF & ~ZF	Above (unsigned)
jb	CF	Below (unsigned)

- ▶ Assemblercode enthält je einen Maschinenbefehl pro Zeile
- ▶ normale Programmausführung ist sequenziell
- ▶ Befehle beginnen an eindeutig bestimmten Speicheradressen
- ▶ **Label**: symbolische Namen für bestimmte Adressen
 - ▶ am Beginn einer Zeile oder vor einem Befehl
 - ▶ vom Programmierer / Compiler vergeben
 - ▶ als **symbolische Adressen** für Sprünge verwendet
- ▶ `_max`: global, Beginn der Funktion `max()`
- ▶ `L9`: lokal, nur vom Assembler verwendete interne Adresse
- ▶ Label müssen in einem Programm eindeutig sein

if-Verzweigung / bedingter Sprung

```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:       # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rax	Rückgabewert

if Übersetzung – goto

- ▶ entspricht C Code mit goto

```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

- ▶ Compilerabhängigkeit

-fno-if-conversion

if Übersetzung – goto (cont.)

- ▶ getrennte Code Abschnitte: then, else
- ▶ „passenden“ ausführen
- ▶ Codeäquivalent

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

```
ntest = !Test;  
if (ntest) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

if Übersetzung – conditional move

- ▶ `cmov..`-Befehl
- ▶ keine Sprünge mehr $\Rightarrow$ gut für Pipelining

-fif-conversion

```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
absdiff:
    movq    %rdi, %rax    # x
    subq    %rsi, %rax    # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx    # eval = y-x
    cmpq    %rsi, %rdi    # x:y
    cmovle  %rdx, %rax    # if <=, result = eval
    ret
```

Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rax	Rückgabewert

if Übersetzung – *conditional move* (cont.)

- ▶ *beide Ausdrücke werden berechnet*
 - + Parallelisierung durch Hardware möglich (*Superskalarität*)
 - Performanz bei komplizierter Berechnung
 - Unsicher, da Seiteneffekte!
- ▶ Codeäquivalent

```
val = Test  
  ? Then_Expr  
  : Else_Expr;
```

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

- ▶ `while...`, `until...`, `for...` Schleifen
 - ▶ können durch Kombinationen von Sprüngen ersetzt werden
 - ▶ bedingte und unbedingte Vorwärts-/Rückwärtssprünge
- ▶ `case...` Mehrfachverzweigungen
 - ▶ Bedingung wird in Zahl umgesetzt
 - ▶ entspricht Offset in Tabelle mit Sprungzielen
 - ▶ dann unbedingter Sprung
- ▶ jetzt weiter mit Unterprogrammen...

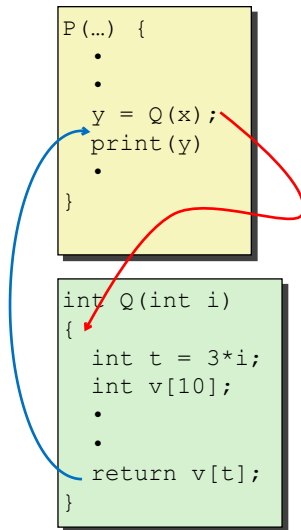
- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für Rückgabewert
- ▶ Speicherverwaltung
 - ▶ Allokation während der Ausführung
 - ▶ Freigabe nach return

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

Mechanismen für Prozeduraufrufe

- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für Rückgabewert
- ▶ Speicherverwaltung
 - ▶ Allokation während der Ausführung
 - ▶ Freigabe nach return



- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für Rückgabewert
- ▶ Speicherverwaltung
 - ▶ Allokation während der Ausführung
 - ▶ Freigabe nach return

```
P (...) {  
  .  
  .  
  y = Q(x);  
  print(y)  
  .  
}
```

```
int Q(int i)  
{  
  int t = 3*i;  
  int v[10];  
  .  
  .  
  return v[t];  
}
```

Mechanismen für Prozeduraufrufe

- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für Rückgabewert
- ▶ Speicherverwaltung
 - ▶ Allokation während der Ausführung
 - ▶ Freigabe nach return

```
P(...) {  
    .  
    .  
    y = Q(x);  
    print(y)  
    .  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    .  
    .  
    return v[t];  
}
```

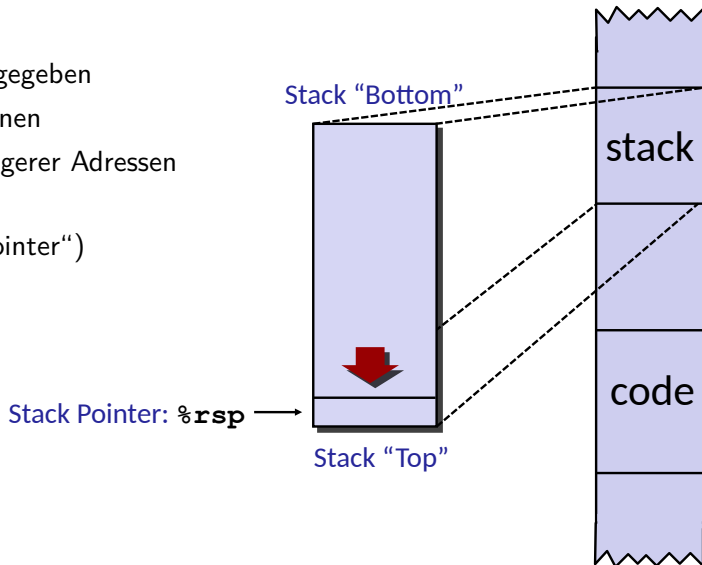
- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für Rückgabewert
- ▶ Speicherverwaltung
 - ▶ Allokation während der Ausführung
 - ▶ Freigabe nach return

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

Stack (Kellerspeicher)

- ▶ Speicherregion
- ▶ Startadresse vom OS vorgegeben
- ▶ Zugriff mit Stackoperationen
- ▶ wächst in Richtung niedrigerer Adressen
- ▶ Register `%rsp` („Stack-Pointer“)
 - ▶ aktuelle Stack-Adresse
 - ▶ oberstes Element



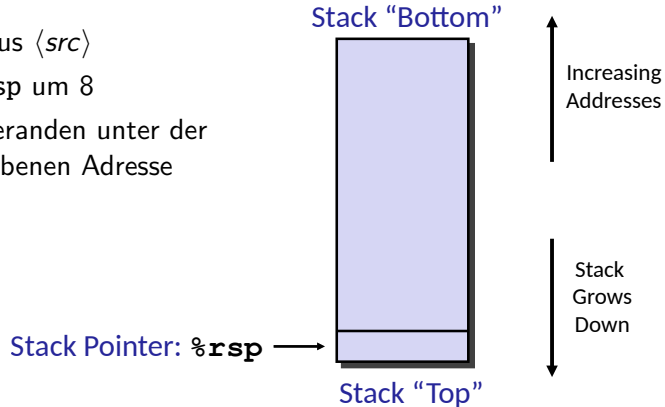
Stack (Kellerspeicher) (cont.)

- ▶ Implementierung von Funktionen/Prozeduren
 - ▶ Speicherplatz für Aufruf-Parameter
 - ▶ Speicherplatz für lokale Variablen
 - ▶ Rückgabe der Funktionswerte
 - ▶ für rekursive Funktionen benötigt!
- ▶ mehrere Varianten/Konventionen
 - ▶ Parameterübergabe in Registern
 - ▶ „Caller-Save“
 - ▶ „Callee-Save“
 - ▶ Kombinationen davon
 - ▶ Aufruf einer Funktion muss deren Konvention berücksichtigen

Stack: Push

`pushq <src>`

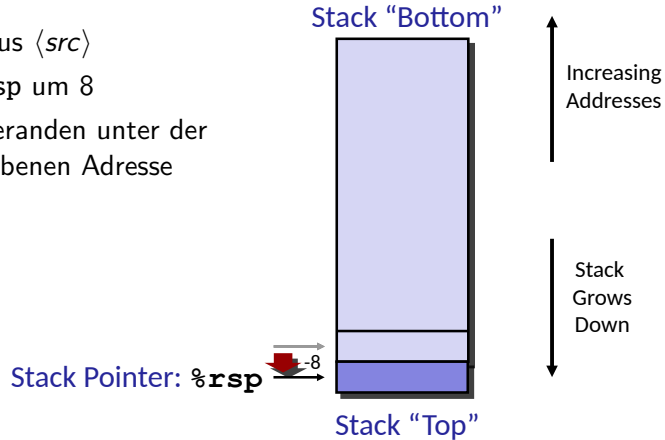
- ▶ holt Operanden aus `<src>`
- ▶ dekrementiert `%rsp` um 8
- ▶ speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Push

`pushq <src>`

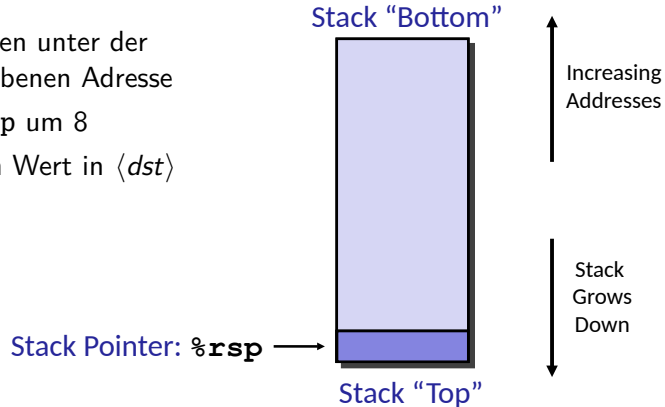
- ▶ holt Operanden aus `<src>`
- ▶ dekrementiert `%rsp` um 8
- ▶ speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Pop

`popq <dst>`

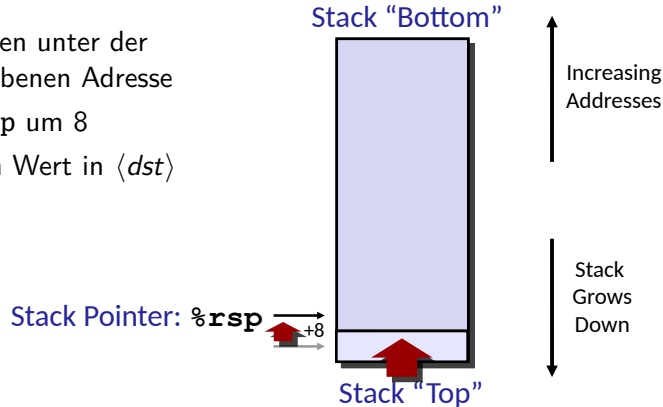
- ▶ liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- ▶ inkrementiert `%rsp` um 8
- ▶ schreibt gelesenen Wert in `<dst>`



Stack: Pop

`popq <dst>`

- ▶ liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- ▶ inkrementiert `%rsp` um 8
- ▶ schreibt gelesenen Wert in `<dst>`



- ▶ x86 ist CISC: spezielle Maschinenbefehle für Funktionsaufruf
 - ▶ `call` zum Aufruf einer Funktion
 - ▶ `ret` zum Rücksprung aus der Funktion
 - ▶ beide Funktionen ähnlich `jmp`: `rip` wird modifiziert
 - ▶ Parameterübergabe über Register und/oder Stack
- ▶ Register mit Spezialaufgaben
 - ▶ `%rsp` „stack-pointer“: Speicheradresse des top-of-stack
 - ▶ `%rbp` „base-pointer“: Speicheradresse des aktuellen Frame
- ▶ Prozeduraufruf: `call <label>`
 - ▶ Rücksprungadresse auf Stack („Push“)
 - ▶ Sprung zu `<label>`
- ▶ Rücksprung: `ret`
 - ▶ Rücksprungadresse vom Stack („Pop“)
 - ▶ Sprung zu dieser Adresse

- ▶ Sprungadressen
 - ▶ Unterprogramm: Adresse der ersten Programmanweisung $\langle label \rangle$
 - ▶ Rücksprung: Adresse der auf den `call` folgenden Anweisung
- ▶ Stack zur Unterstützung von `call` und `ret`
 - ▶ ggf. Parameter
 - ▶ Rücksprungadresse
 - ▶ lokale Variablen
 - ▶ ggf. Rückgabewerte

Codebeispiel Unterprogrammaufruf

```
void multstore(long x, long y, long
*dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
400540: push    %rbx          # Save %rbx
400541: mov     %rdx,%rbx      # Save dest
400544: callq   400550 <mult2> # mult2(x,y)
400549: mov     %rax,(%rbx)     # Save at dest
40054c: pop     %rbx           # Restore %rbx
40054d: retq                    # Return
```

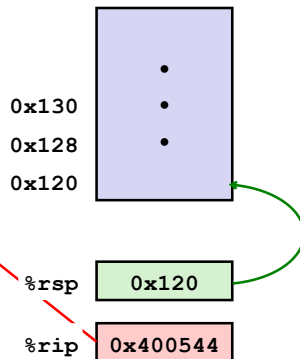
```
long mult2(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
400550: mov     %rdi,%rax      # a
400553: imul    %rsi,%rax      # a * b
400557: retq                    # Return
```

► Prozeduraufruf callq

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov  %rax, (%rbx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: mov  %rdi,%rax  
.  
.  
400557: retq
```

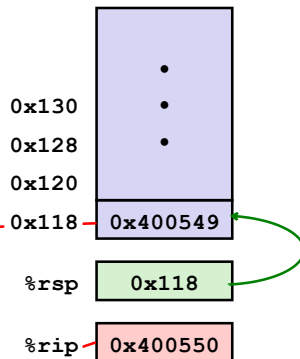


Kontrollübergabe (cont.)

- ▶ Rücksprungadresse auf Stack
- ▶ Programmzähler setzen %rip

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov   %rax, (%rbx)  
.  
.
```

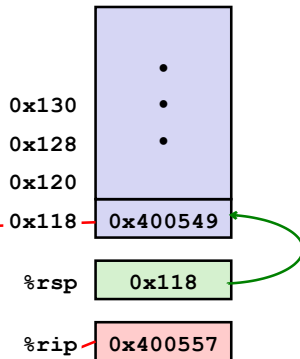
```
0000000000400550 <mult2>:  
400550: mov   %rdi,%rax  
.  
.  
400557: retq
```



► Rücksprung `retq`

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov  %rax, (%rbx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: mov  %rdi,%rax  
.  
.  
400557: retq
```

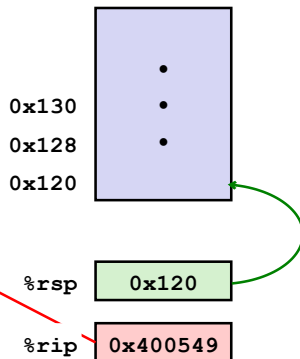


Kontrollübergabe (cont.)

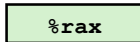
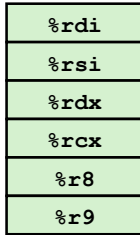
- ▶ Rücksprungadresse vom Stack
- ▶ Programmzähler setzen %rip

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov   %rax, (%rbx)  
.  
.
```

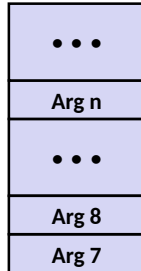
```
0000000000400550 <mult2>:  
400550: mov   %rdi,%rax  
.  
.  
400557: retq
```



► Register



Stack



► Konvention

- die ersten 6 Argumente: Register `%rdi`, `%rsi`...
- Stack wenn notwendig
- Rückgabewert: Register `%rax`

Datenübergabe (cont.)

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    . . .
400541: mov     %rdx,%rbx      # Save dest
400544: callq   400550 <mult2>  # mult2(x,y)
    # t in %rax
400549: mov     %rax,(%rbx)     # Save at dest
    . . .
```

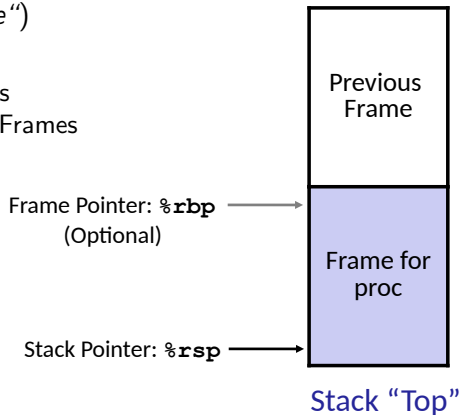
```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
400550: mov     %rdi,%rax      # a
400553: imul    %rsi,%rax      # a * b
    # s in %rax
400557: retq                               # Return
```

- ▶ für alle Programmiersprachen, die Rekursion unterstützen
 - ▶ C, Pascal, Java, Lisp usw.
 - ▶ Code muss „reentrant“ sein
 - ▶ erlaubt mehrfache, simultane Instanziierungen einer Prozedur
 - ▶ benötigt Platz, um den Zustand jeder Instanziierung zu speichern
 - ▶ ggf. Argumente
 - ▶ lokale Variable(n)
 - ▶ Rücksprungsadresse
- ▶ Stack-„Prinzip“
 - ▶ dynamischer Zustandsspeicher für Aufrufe
 - ▶ zeitlich limitiert: vom Aufruf (`call`) bis zum Rücksprung (`ret`)
 - ▶ aufgerufenes Unterprogramm („*Callee*“) wird vor dem aufrufenden Programm („*Caller*“) beendet
- ▶ Stack-„Frame“
 - ▶ der Bereich/Zustand einer einzelnen Prozedur-Instanziierung

- ▶ alle Daten für einen Funktionsaufruf („*Closure*“)
- ▶ Adressverweise („*Pointer*“)
 - ▶ Stackpointer `%rsp`: das obere Ende des Stacks
 - ▶ Framepointer `%rbp`: der Anfang des aktuellen Frames

- ▶ Daten
 - ▶ ggf. Aufruf-Parameter
 - ▶ Rücksprungadresse
 - ▶ ggf. lokale Variablen
 - ▶ ggf. temporäre Daten



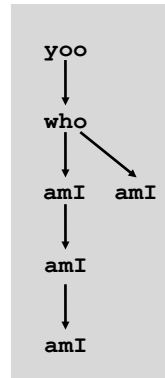
- ▶ Speicherverwaltung durch Funktion
 - ▶ bei Aufruf wird Stack gefüllt: „Set-up“ Code
 - ▶ bei Return wieder freigegeben: „Finish“ Code

Beispiel: Prozeduraufrufe

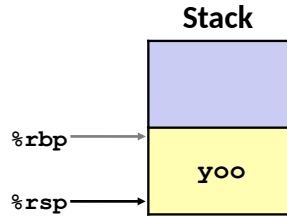
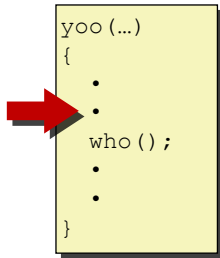
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI ();  
  . . .  
  amI ();  
  . . .  
}
```

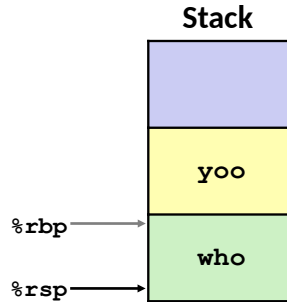
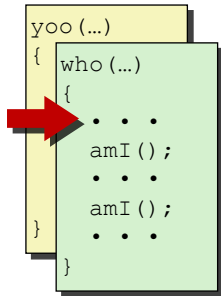
```
amI (...)  
{  
  .  
  .  
  amI ();  
  .  
  .  
}
```



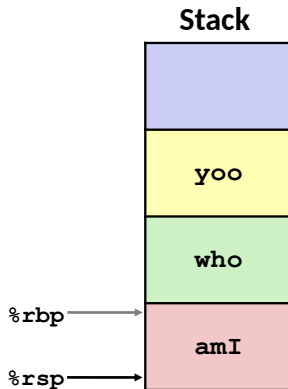
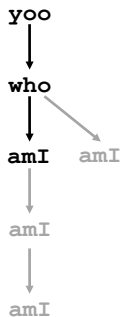
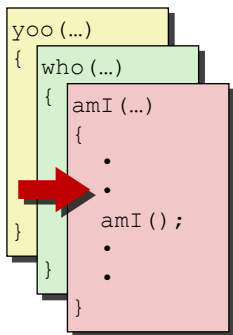
Beispiel: Prozeduraufrufe (cont.)



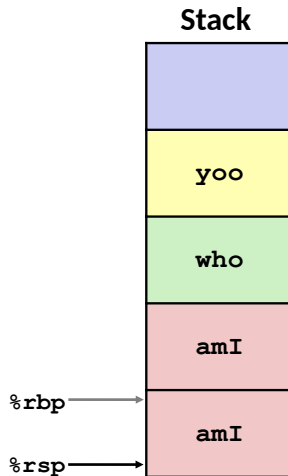
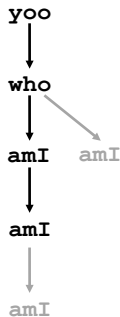
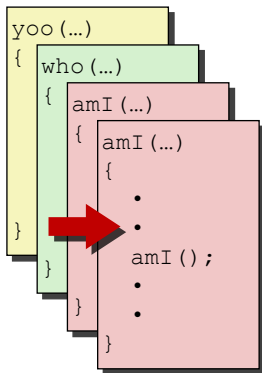
Beispiel: Prozeduraufrufe (cont.)



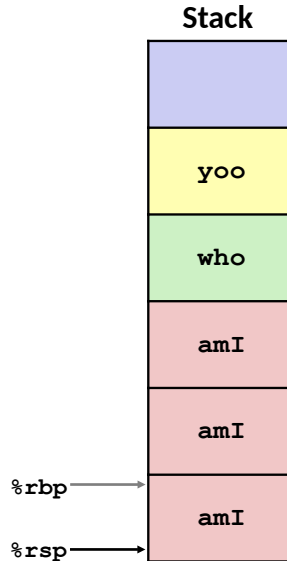
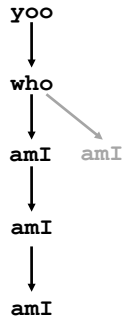
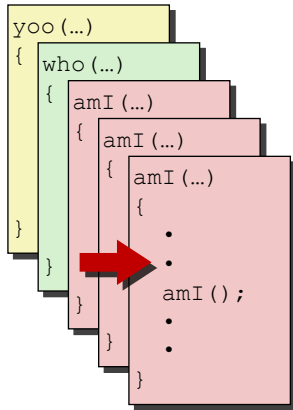
Beispiel: Prozeduraufrufe (cont.)



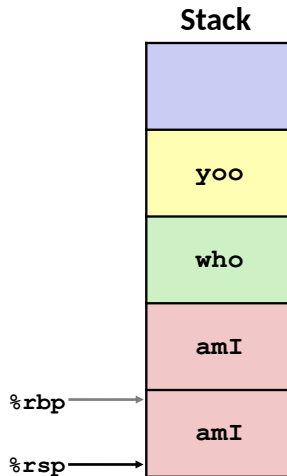
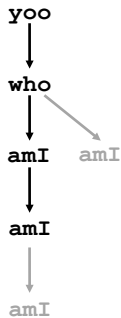
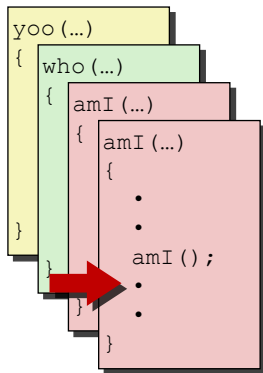
Beispiel: Prozeduraufrufe (cont.)



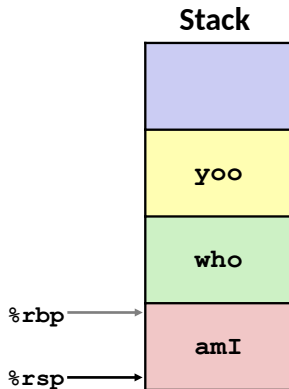
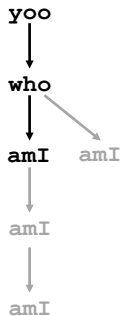
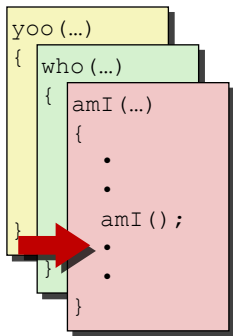
Beispiel: Prozeduraufrufe (cont.)



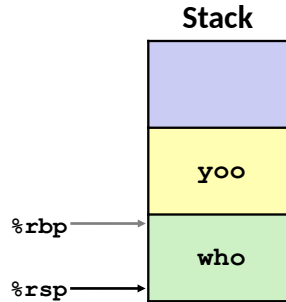
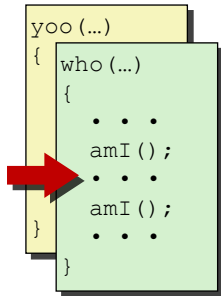
Beispiel: Prozeduraufrufe (cont.)



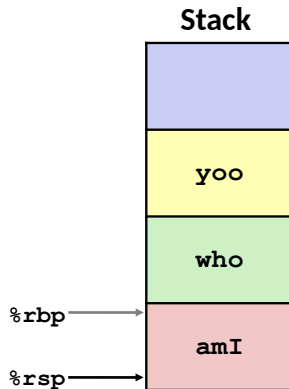
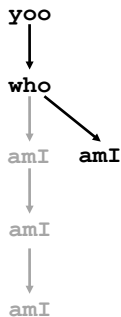
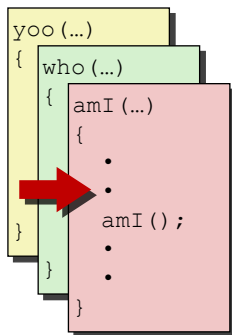
Beispiel: Prozeduraufrufe (cont.)



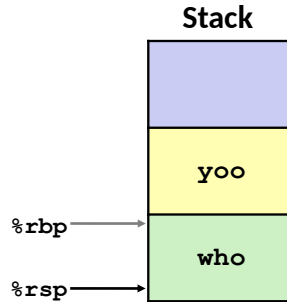
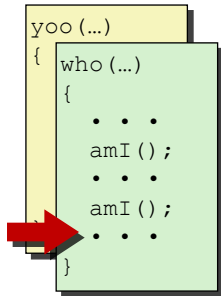
Beispiel: Prozeduraufrufe (cont.)



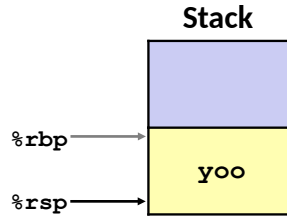
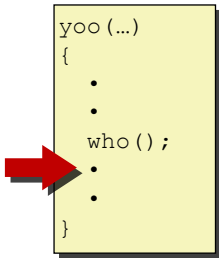
Beispiel: Prozeduraufrufe (cont.)



Beispiel: Prozeduraufrufe (cont.)



Beispiel: Prozeduraufrufe (cont.)

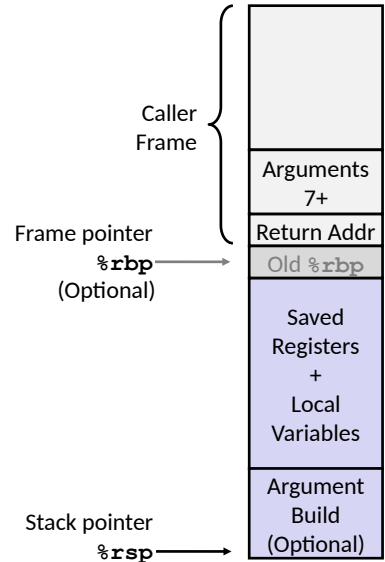


aktueller Stack-Frame / „Callee“

- ▶ Argumente: Parameter für Funktionsaufruf
- ▶ lokale Variablen
 - ▶ wenn sie nicht in Registern gehalten werden können
- ▶ gespeicherter Registerkontext
- ▶ Zeiger auf vorherigen Frame

„Caller“ Stack-Frame

- ▶ Rücksprungadresse
 - ▶ von `call`-Anweisung erzeugt
- ▶ Argumente für aktuellen Aufruf



- ▶ `yoo` („Caller“) ruft Prozedur `who` („Callee“) auf

⇒ *Welche Register können temporär von `who` genutzt werden?*

- ▶ zwei mögliche Konventionen
 - ▶ „Caller-Save“: `yoo` speichert in seinen Frame vor Prozeduraufruf
 - ▶ „Callee-Save“: `who` speichert in seinen Frame vor Benutzung

Register Sicherungskonventionen (cont.)

▶ **%rax**

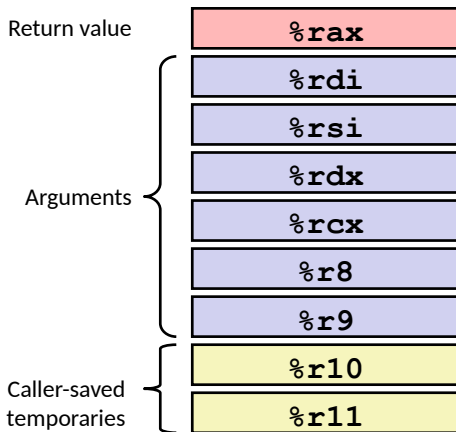
- ▶ Rückgabewert
- ▶ Caller-Save
- ▶ kann lokal geschrieben werden

▶ **%rdi...%r9**

- ▶ Argumente
- ▶ Caller-Save
- ▶ können lokal geschrieben werden

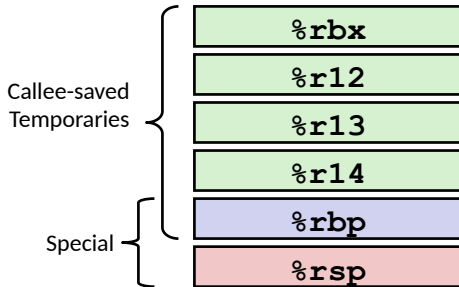
▶ **%r10, %r11**

- ▶ Caller-Save
- ▶ können lokal geschrieben werden



Register Sicherungskonventionen (cont.)

- ▶ **%rbx, %r12...%r14**
 - ▶ Callee-Save
 - ▶ Prozedur muss sichern (Stack-Frame) und zurückschreiben
- ▶ **%rbp**
 - ▶ Callee-Save
 - ▶ Prozedur muss sichern (Stack-Frame) und zurückschreiben
 - ▶ Frame-Pointer $\hat{=}$ Beginn des eigenen Frames
- ▶ **%rsp**
 - ▶ Behandlung durch `call/return`
 - ▶ Sonderfall, quasi Callee-Save



- Programm in mehrere Quelldateien aufgeteilt

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

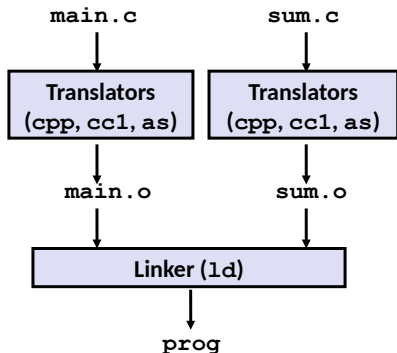
```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

- ▶ Compiler(-driver) startet einzelne Programme
 - ▶ Präprozessor (cpp), Compiler (cc), Assembler (as) und Linker (ld)
 - ▶ „Feintuning“ und Steuerung über Kommandozeilen-Parameter
'zig Optionen für jedes Teilprogramm (s. man gcc)

Linux gcc



+ Modularität

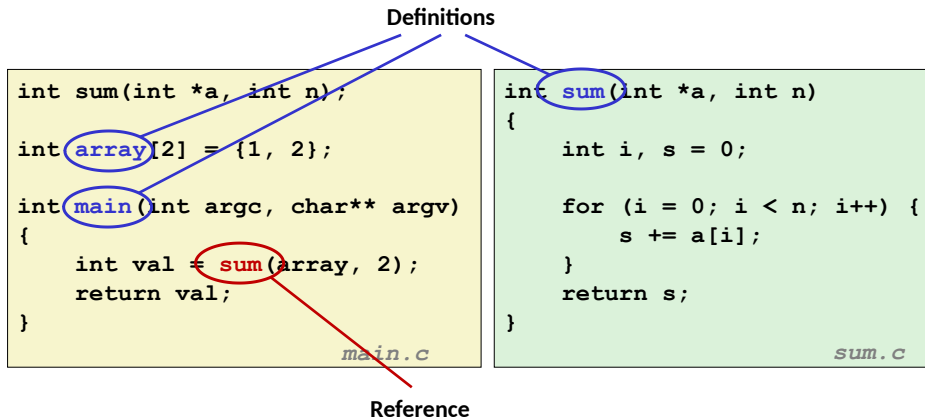
- ▶ Programm in übersichtlichen kleinen Dateien
 - ▶ Funktionen können wiederverwendet werden
- ⇒ vorgefertigte Programmbibliotheken

+ Effizienz

- ⇒ Zeitvorteil
- ▶ nach Änderung müssen nur kleine Teile neu übersetzt werden
 - ▶ ermöglicht paralleles Compilieren
- ⇒ (Speicher-) Platzvorteil
- ▶ wichtige Funktionen in Programmbibliotheken aggregiert
 - ▶ ermöglicht gemeinsame Nutzung ⇒ *Shared Libraries*
 - ▶ effizienteres, vereinfachtes Programmieren

1. Symbole identifizieren (globale Variablen, Funktionen)

Symbole auflösen (eindeutig machen) $\Rightarrow$ Symboltabelle

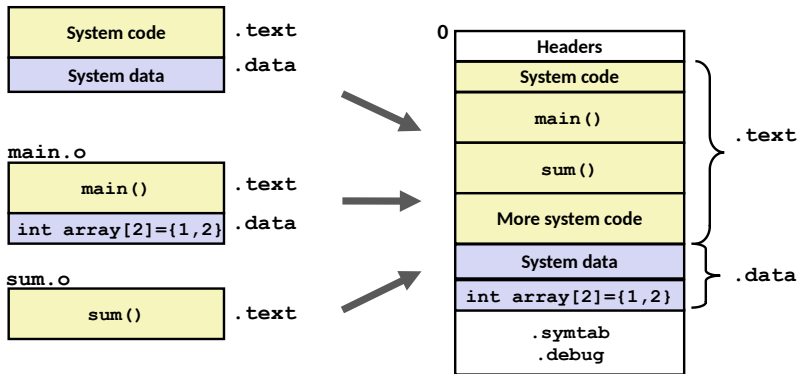


Aufgaben des Linkers (cont.)

2. „Relocation“

- ▶ Programmcode und -daten der Quelldateien zusammenführen
- ▶ alle Symboltabellen zusammenfassen

⇒ gemeinsamer, eindeutiger Adressraum: Sprünge+Symbole



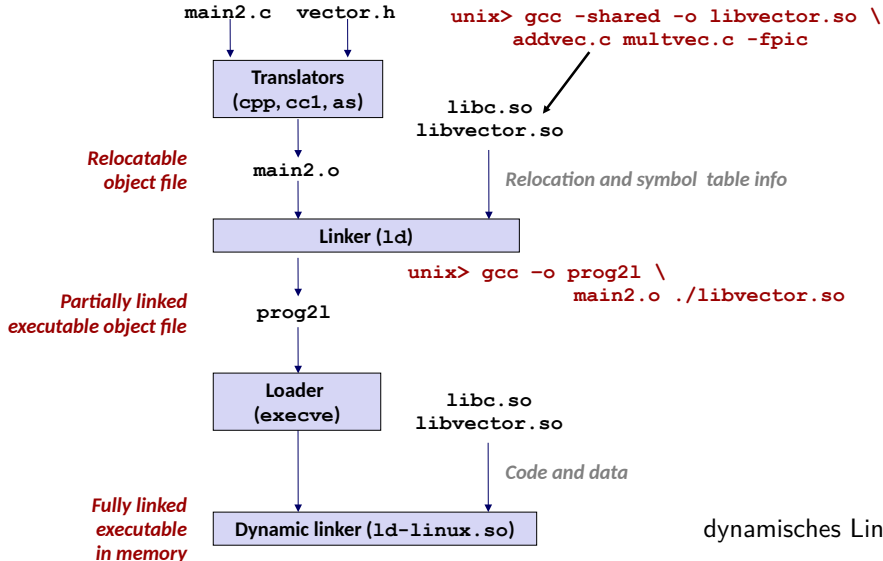
⇒ erzeugt ein ausführbares Programm für *Loader*

- ▶ **Statisches Binden** („static linking“)
 - ▶ Funktionen aus Bibliotheksarchiven (.a-Dateien) werden in das Programm integriert
 - ▶ nicht genutzte Funktionen werden entfernt
 - ▶ Linken während Compilierung

- ▶ **Dynamisches Binden** („dynamic linking“)
 - ▶ Bibliotheken werden erst beim Laden in Speicher oder erst zur Laufzeit dazugelinkt
 - ▶ gemeinsame Nutzung von Funktionen durch mehrere Prozesse (incl. Betriebssystem); die zugehörigen Bibliotheken liegen aber (maximal) einmal im Speicher

 - ▶ signifikant effizienter als separat statische gelinkte Programme
 - ▶ Linux: .so-Dateien – „Shared Object“
Windows: .dll-Dateien – „Dynamic Link Libraries“

Statisches / dynamisches Linken (cont.)



dynamisches Linken beim Laden

viele Themen aus Zeitgründen nicht behandelt

- ▶ Linker und Loader
 - ▶ genauere Funktionsweise von Linker und Loader
 - ▶ programmiertechnische Realisierung
 - ▶ Probleme bei der Symbolauflösung
- ▶ Speicherverwaltung
 - ▶ Abbildung der Datenstrukturen auf Bytes im Speicher
 - ▶ Adressberechnung für Arrays, Records
 - ▶ Alignment
 - ▶ dynamische Speicherverwaltung / der „Heap“
- ▶ Objektorientierte Konzepte
 - ▶ Daten mit zugehörigen Methoden kombinieren

- ▶ *Was kann zur Laufzeit alles schief gehen?*
 - ▶ Pufferüberläufe
 - ▶ Sicherheitsaspekte
- ▶ *Wie ist die Verbindung zum Betriebssystem?*
- ▶ ...

weitere Informationen unter:

- R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective [BO15]
- die „passende“ Vorlesung der *Carnegie Mellon Uni.*
www.cs.cmu.edu/~213 – Foliensätze unter „Schedule“

- [BO15] Randal E. Bryant and David R. O'Hallaron.
Computer systems – A programmers perspective.
Pearson Education Limited, Harlow, third, global ed. edition, 2015.
- [Hyd10] Randall Hyde.
The Art of Assembly Language.
No Starch Press, San Francisco, CA, second edition, 2010.
- [Hyd21] Randall Hyde.
The Art of 64-bit Assembly, Volume 1.
No Starch Press, San Francisco, CA, 2021.
- [IA6] Intel Corp., Santa Clara, CA.
Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture.

- [PH20] David A. Patterson and John L. Hennessy.
Computer Organization and Design – The Hardware Software Interface – RISC-V Edition.
Morgan Kaufmann Publishers Inc., San Mateo, CA, second edition, 2020.
- [PH22] David A. Patterson and John L. Hennessy.
Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition.
De Gruyter Oldenbourg, Berlin, sixth edition, 2022.
- [TA14] Andrew S. Tanenbaum and Todd Austin.
Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.
Pearson Deutschland GmbH, Hallbergmoos, sixth edition, 2014.