64-040 Modul InfB-RSB
Rechnerstrukturen und Betriebssysteme

https://tams.informatik.uni-hamburg.de/lectures/2024ws/vorlesung/rsb

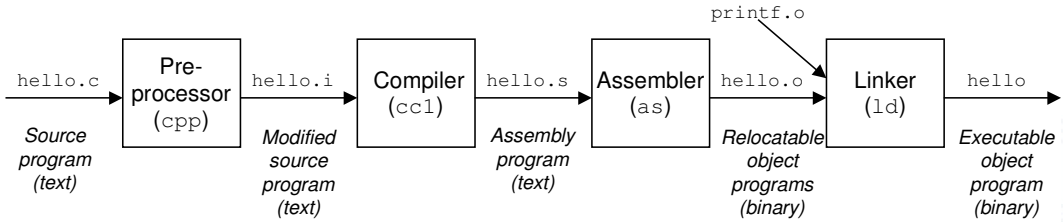
– Kapitel 13 –

Andreas Mäder

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

Wintersemester 2024/2025

Wdh. Compilierungssystem



[BO15]

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

Warum Assembler?

Programme werden nur noch selten in Assembler geschrieben

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

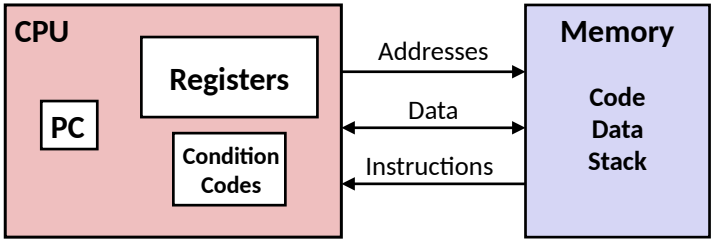
aber Grundwissen bleibt trotzdem unverzichtbar

- 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

Assembler in RSB

- 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.)
  - ...

Assemblersicht des Programmierers



[BO15]

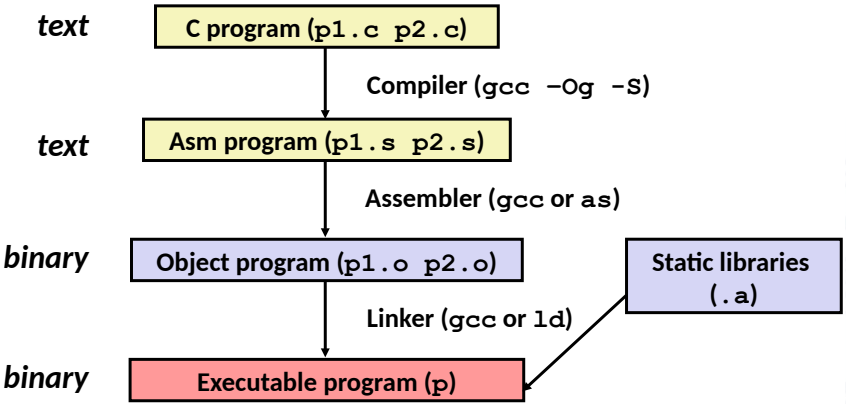
beobachtbare Zustände

- Programmzähler (*Instruction Pointer*) — x86-64 `rip` Register
  - Adresse der nächsten Anweisung
- Registerbank — `rax` . . . `rbp` Register
  - häufig benutzte Programmdaten — `r8` . . . `r15` Register
- Zustandscodes — EFLAGS Register
  - Statusinformationen über die letzte ALU Operation
  - für bedingte Sprünge benötigt (*Conditional Branch*)

Assemblersicht des Programmierers (cont.)

- Speicher
  - byteweise adressierbares Array
  - pro Task: Code, Nutzerdaten, (einige) OS Daten
  - –"– Kellerspeicher für Unterprogrammaufrufe „Stack“
  - –"– Adressraum dynamisch alloziierter 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
```

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
```

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

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

Assembler: Charakteristika

- 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

Assembler: Datentypen

- 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	8 . . . 64 bits
Datenwerte für Variablen	int/long/long long
positiv oder vorzeichenbehaftet	signed/unsigned
Textzeichen (ASCII, Unicode)	char
Gleitkomma-Daten mit 4 oder 8 Bytes	float/double
Adressen bzw. „Pointer“	untypisierte Speicheradressen

Assembler: Befehle/Operationen

- 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 und Linker

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 `malloc, printf`
- 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`

Objektcode Disassembler: objdump

[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, ausführbaren Programm ⟨*executable*⟩ oder einer `.o` Datei ausgeführt werden

Was kann „disassembliert“ werden?


```
% 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

[BO15]

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

- Adressierungsarten
- arithmetische Operationen
- Statusregister
- Umsetzung von Programmstrukturen

Einschränkungen

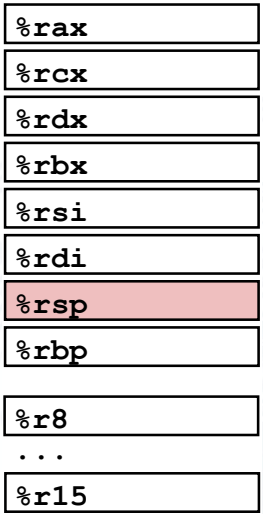
- Beispiele nutzen nur die 64-bit Datentypen
  `long` bei Linux (unter Windows nur 4-Byte!)
  - x86-64 wird wie 16-Register 64-bit Maschine benutzt (=RISC)
  - CISC Komplexität und Tricks bewusst vermieden
- Beispiele nutzen gcc/as Syntax (vs. Microsoft, Intel)

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



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` → `x`
  - positiver (oder negativer) Integerwert
- Register: `R` → `Reg[R]`
  - Inhalt eines der 16 Universalregister `%rax` . . . `%r15`
    Registername `R` beginnt immer mit `%`
- Normal: `(R)` → `Mem[Reg[R]]`
  - Register `R` spezifiziert die Speicheradresse
  - Beispiel: `movq (%rcx), %rax`
- Displacement: `D(R)` → `Mem[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

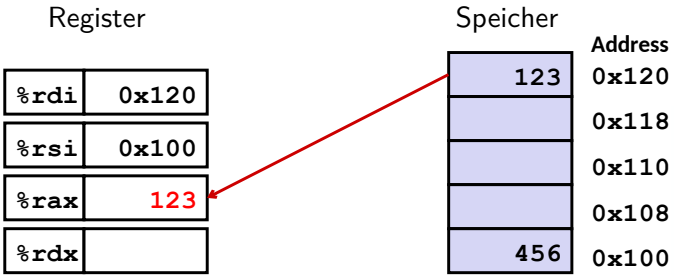
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Speicher

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

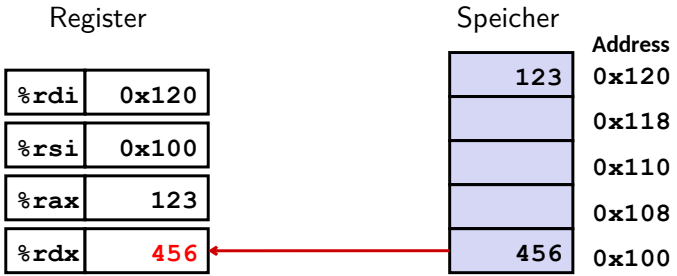
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Funktionsweise von swap()



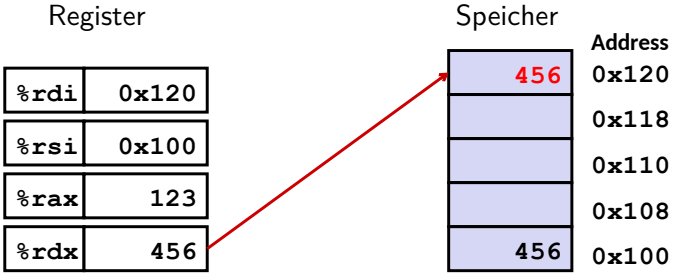

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Funktionsweise von swap()



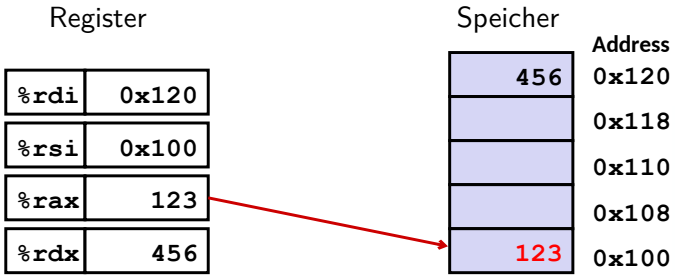

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

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
```

Indizierte Adressierung

- allgemeine Form
  - `Imm(Rb,Ri,S)` → `Mem[Reg[Rb]+S*Reg[Ri]+Imm]`
    - ⟨*Imm*⟩ Offset
    - ⟨*Rb*⟩ Basisregister: eines der 16 Integer-Register
    - ⟨*Ri*⟩ Indexregister: jedes außer `%rsp`
      `%rbp` grundsätzlich möglich, jedoch unwahrscheinlich
    - ⟨*S*⟩ Skalierungsfaktor 1, 2, 4 oder 8

- gebräuchlichste Fälle
  - `(Rb)` → `Mem[Reg[Rb]]`
  - `Imm(Rb)` → `Mem[Reg[Rb] + Imm]`
  - `(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri]]`
  - `Imm(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri] + Imm]`
  - `(Rb,Ri,S)` → `Mem[Reg[Rb] + S*Reg[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`

Arithmetische Operationen

- binäre Operatoren

Format		Berechnung	
`addq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩+⟨*src*⟩	
`subq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩-⟨*src*⟩	
`imulq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩*⟨*src*⟩	
`salq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩<<⟨*src*⟩	auch `shlq`
`sarq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩>>⟨*src*⟩	arithmetisch
`shrq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩>>⟨*src*⟩	logisch
`xorq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩^⟨*src*⟩	
`andq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩&⟨*src*⟩	
`orq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩ | ⟨*src*⟩	

Arithmetische Operationen (cont.)

- unäre Operatoren

Format		Berechnung
`incq`	⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩+1
`decq`	⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩-1
`negq`	⟨*dst*⟩	⟨*dst*⟩ = -⟨*dst*⟩
`notq`	⟨*dst*⟩	⟨*dst*⟩ = ~⟨*dst*⟩

- `leaq`-Befehl: *load effective address*

  `leaq` ⟨*src*⟩ , ⟨*dst*⟩

  - Adressberechnung für (späteren) Ladebefehl
  - Speichert die Adressausdruck ⟨*src*⟩ in Register ⟨*dst*⟩
    `Imm(Rb,Ri,S)` → `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
```

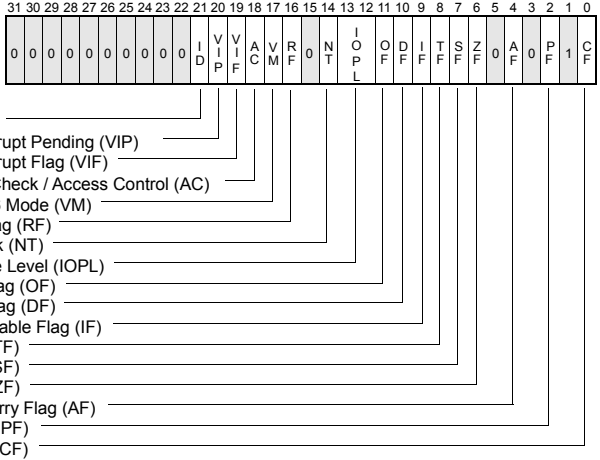
Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z, t4
%rax	t1, t2, rval
%rcx	t5

Kontrollfluss / Programmstrukturen

- Zustandscodes
  - Setzen
  - Testen

- Ablaufsteuerung
  - Verzweigungen: „If-then-else“
  - Schleifen: „Loop“-Varianten
  - Mehrfachverzweigungen: „Switch“

x86: EFLAGS Register

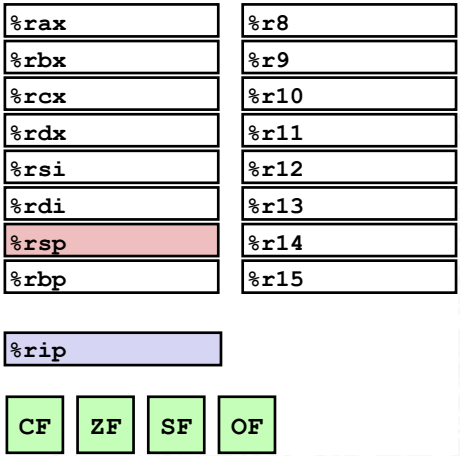


- x86-64: RFLAGS $\widehat{=}$ EFLAGS, mit „0“ erweitert

[IA64]

Prozessor aus Sicht des Programmierers

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



Zustandscodes

- 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` $\langle src \rangle$, $\langle dst \rangle$ 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
     $(a > 0 \;\&\&\; b > 0 \;\&\&\; t < 0) \;||\; (a < 0 \;\&\&\; b < 0 \;\&\&\; t \geq 0)$

Zustandscodes (cont.)

2. explizites Setzen durch Vergleichsoperation
   - Beispiel: `cmpq` $\langle src2 \rangle, \langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle - \langle src2 \rangle$ (`subq` $\langle src2 \rangle, \langle src1 \rangle$)
     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
     $(a > 0 \;\&\&\; b < 0 \;\&\&\; (a - b) < 0) \;||\; (a < 0 \;\&\&\; b > 0 \;\&\&\; (a - b) \geq 0)$

Zustandscodes (cont.)

3. explizites Setzen durch Testanweisung
   - Beispiel: `testq` $\langle src2 \rangle, \langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle \& \langle src2 \rangle$ (`andq` $\langle src2 \rangle, \langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats
   - $\Rightarrow$ 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)

Assembler: Label

- 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 `-fif-conversion`
- keine Sprünge mehr ⇒ gut für Pipelining

```
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;
```

andere Kontrollstrukturen

- `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...

Mechanismen für Prozeduraufrufe

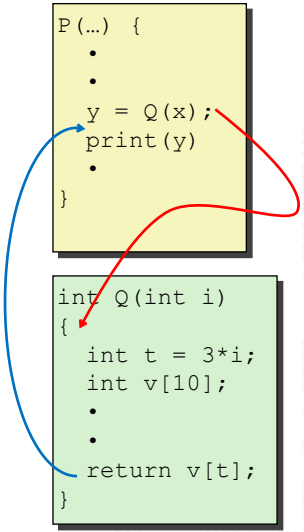
- Kontrollübergabe
  - zum Unterprogrammcode
  - zurück zum Aufruf
- Datenübergabe
  - der Argumente
  - für Rückgabewert
- Speicherverwaltung
  - Allokation während der Ausführung
  - Freigabe nach `return`

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

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

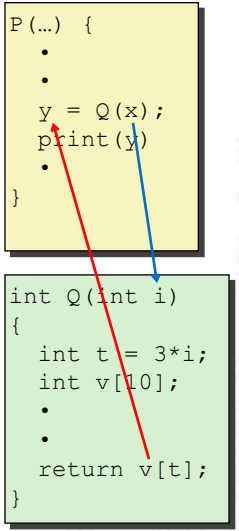
Mechanismen für Prozeduraufrufe

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



Mechanismen für Prozeduraufrufe

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



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];
}
```

Mechanismen für Prozeduraufrufe

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

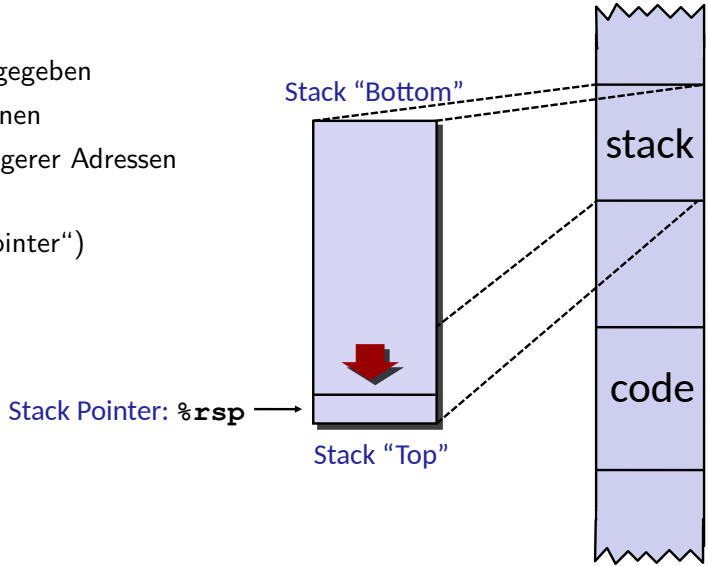
```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

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

Stack (Kellerspeicher)

- Speicherregion
- Startadresse vom OS vorgegeben
- Zugriff mit Stackoperationen
- wächst in Richtung niedrigerer Adressen

- Register `%rsp` („Stack-Pointer“)
  - aktuelle Stack-Adresse
  - oberstes Element



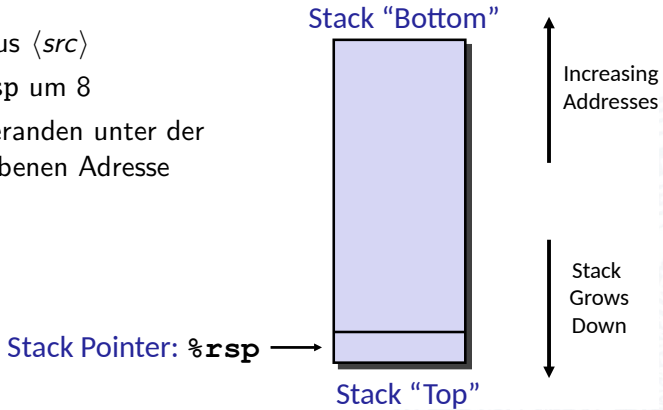
Stack (Kellerspeicher) (cont.)

- 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*⟩

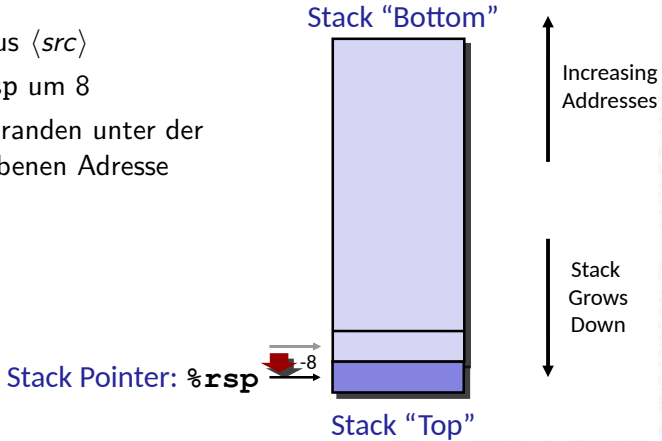
- holt Operanden aus ⟨*src*⟩
- dekrementiert `%rsp` um 8
- speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Push

`pushq` ⟨*src*⟩

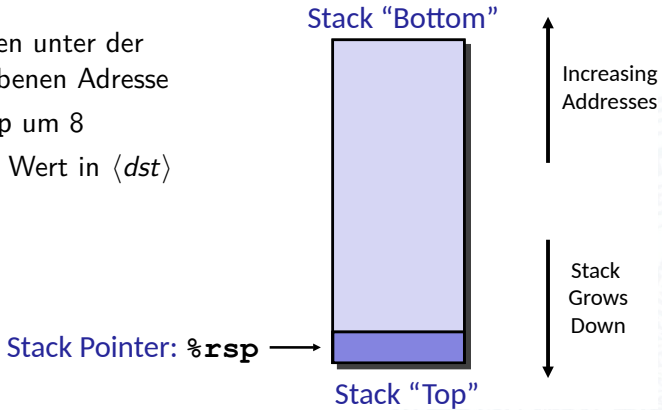
- holt Operanden aus ⟨*src*⟩
- dekrementiert `%rsp` um 8
- speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Pop

`popq` ⟨*dst*⟩

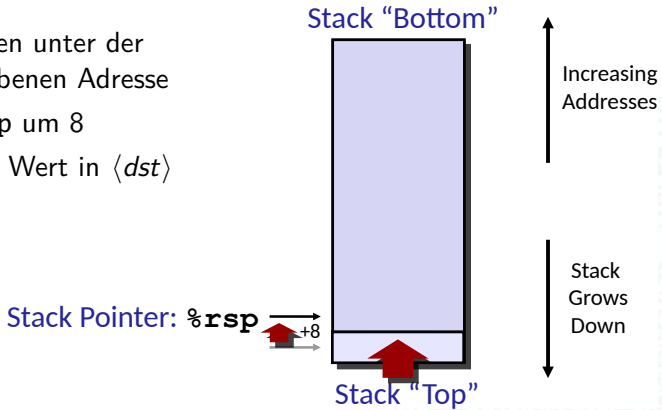
- liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- inkrementiert `%rsp` um 8
- schreibt gelesenen Wert in ⟨*dst*⟩



Stack: Pop

`popq` ⟨*dst*⟩

- liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- inkrementiert `%rsp` um 8
- schreibt gelesenen Wert in ⟨*dst*⟩



Funktions-/Prozeduraufruf

- 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

Funktions-/Prozeduraufruf (cont.)

- Sprungadressen
  - Unterprogramm: Adresse der ersten Programmanweisung ⟨*label*⟩
  - 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
```

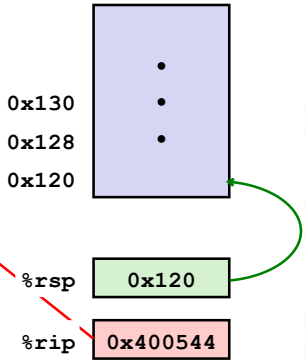
```
long mult2(long a, long b)
{
  long s = a * b;
  return s;
}
```

```
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)
  •
  •
```

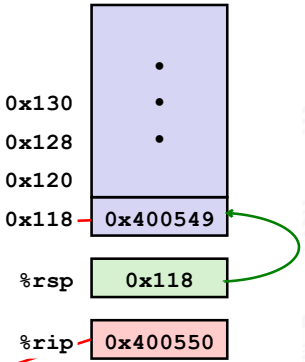
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Kontrollübergabe (cont.)

- Rücksprungadresse auf Stack
- Programmzähler setzen `%rip`

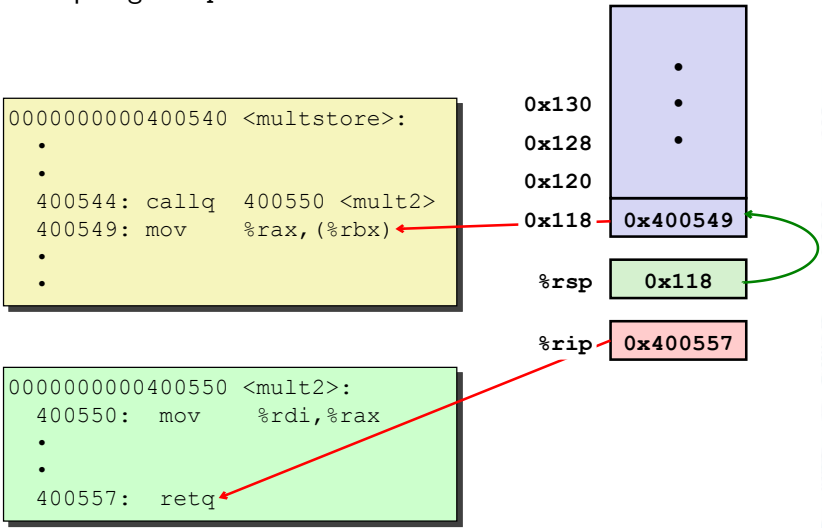
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```




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

Kontrollübergabe (cont.)

- Rücksprung `retq`

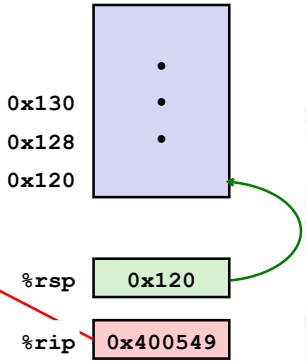


Kontrollübergabe (cont.)

- Rücksprungadresse vom Stack
- Programmzähler setzen `%rip`

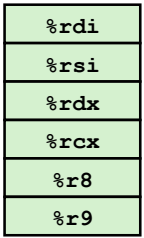
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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

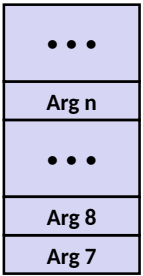


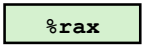
Datenübergabe

- 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
```

Stack-basierende Programmierung

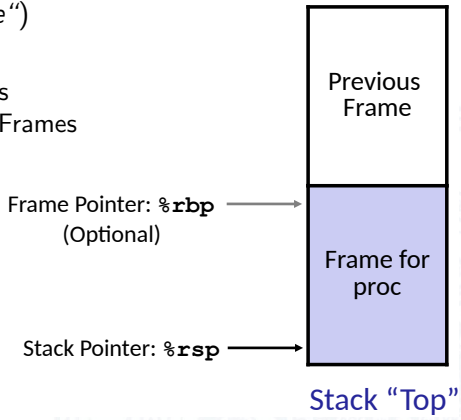
- für alle Programmiersprachen, die Rekursion unterstützen
  - C, Pascal, Java, Lisp usw.
    - Code muss „reentrant“ sein
    - erlaubt mehrfache, simultane Instanziierungen einer Prozedur
  - benötigt Platz, um den Zustand jeder Instanziierung zu speichern
    - ggf. Argumente
    - lokale Variable(n)
    - Rücksprungadresse
- 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

Stack-Frame

- 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

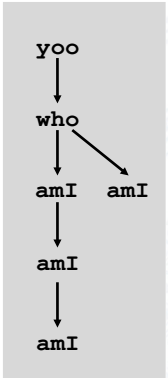


Beispiel: Prozeduraufrufe

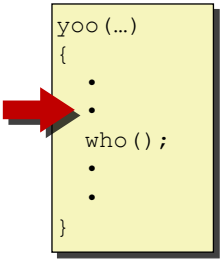

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

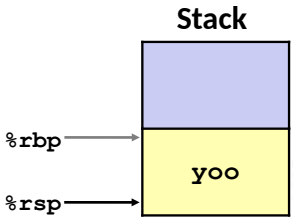
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

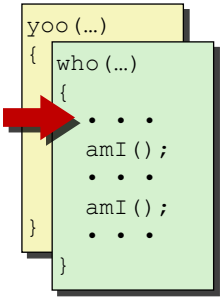


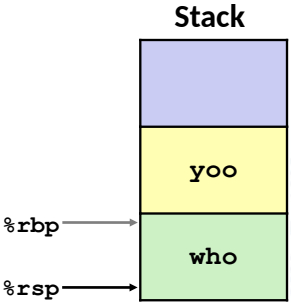
Beispiel: Prozeduraufrufe (cont.)



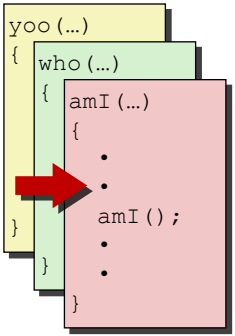


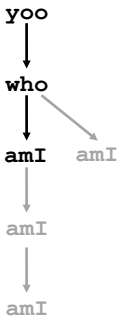
Beispiel: Prozeduraufrufe (cont.)

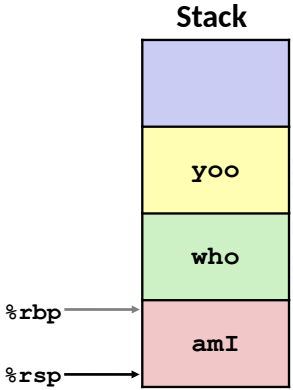




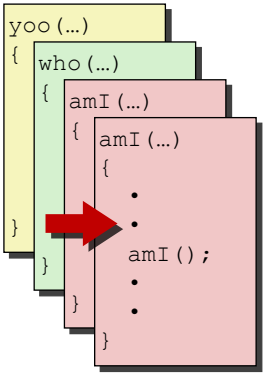
Beispiel: Prozeduraufrufe (cont.)

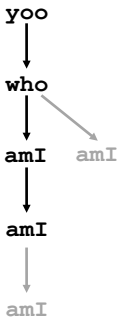


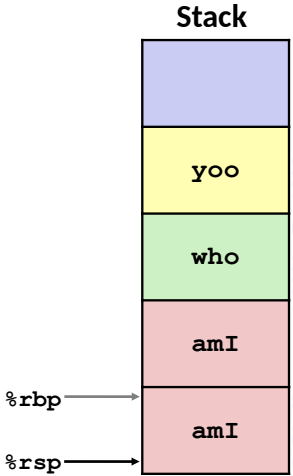




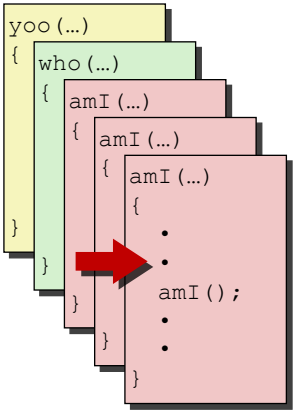
Beispiel: Prozeduraufrufe (cont.)

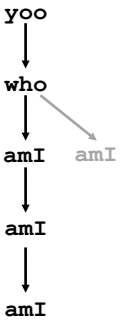


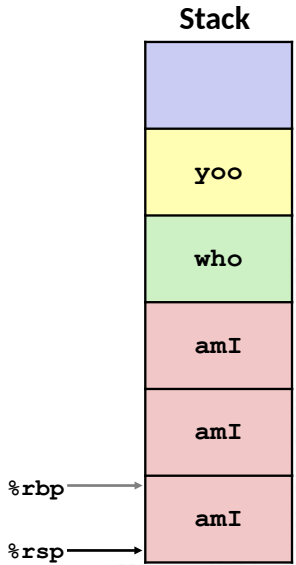




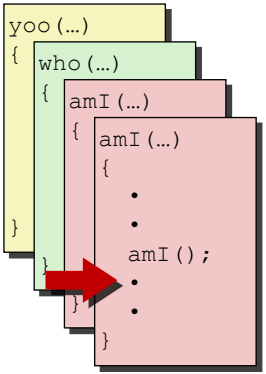
Beispiel: Prozeduraufrufe (cont.)

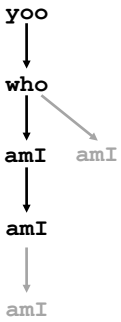


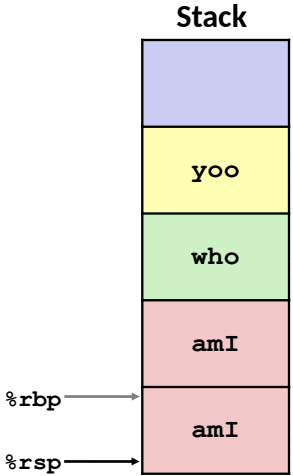




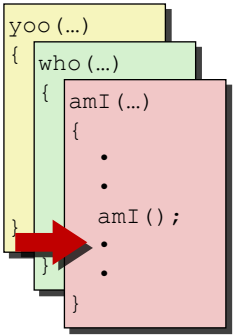
Beispiel: Prozeduraufrufe (cont.)

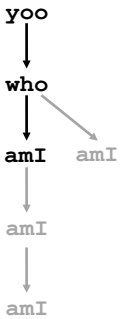


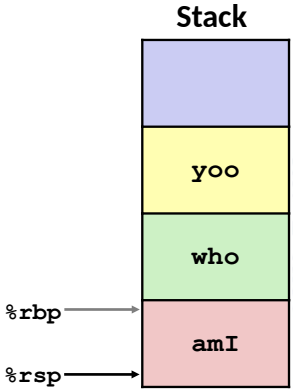




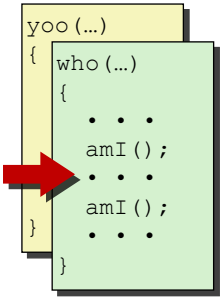
Beispiel: Prozeduraufrufe (cont.)

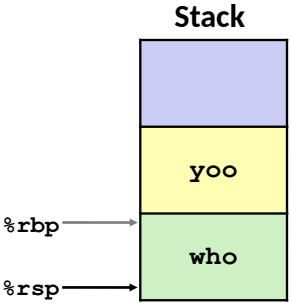




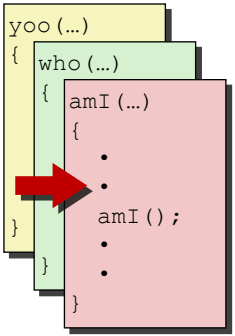


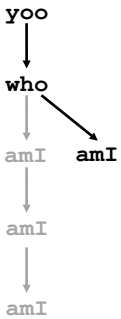
Beispiel: Prozeduraufrufe (cont.)

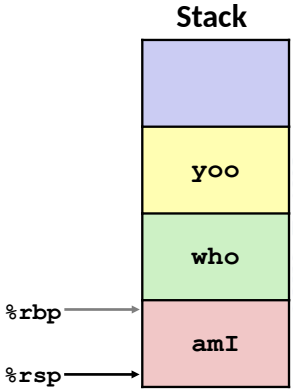




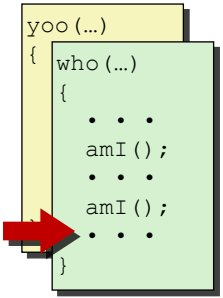
Beispiel: Prozeduraufrufe (cont.)

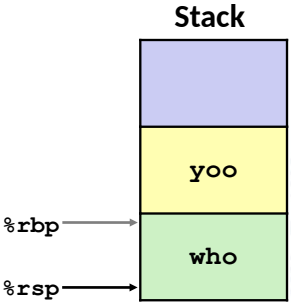




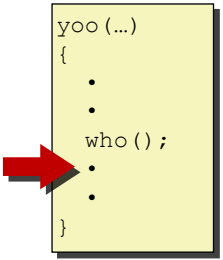


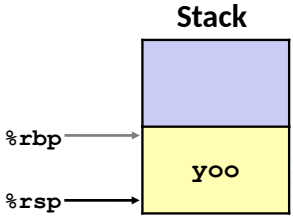
Beispiel: Prozeduraufrufe (cont.)





Beispiel: Prozeduraufrufe (cont.)





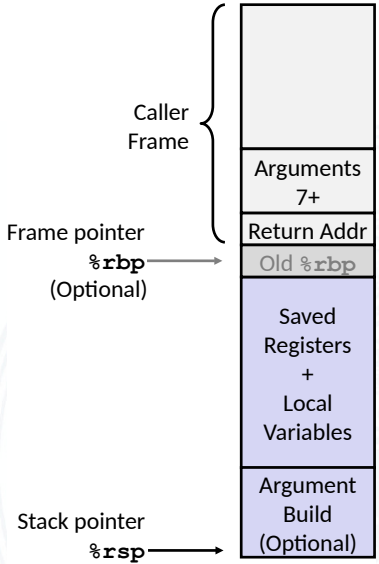
x86-64/Linux Stack-Frame

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



Register Sicherungskonventionen

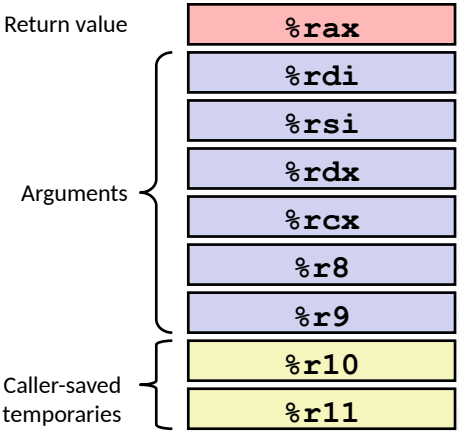
- yoo („Caller“) ruft Prozedur `who` („Callee“) auf

$\Rightarrow$ *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

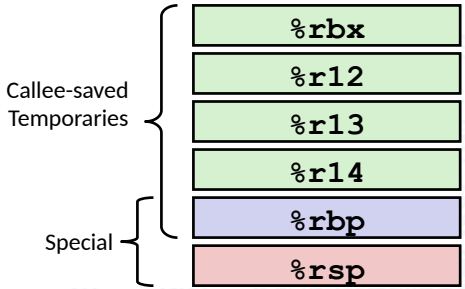
Register Sicherungskonventionen (cont.)

- `%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 $\widehat{=}$ Beginn des eigenen Frames
- `%rsp`
  - Behandlung durch `call`/`return`
  - Sonderfall, quasi Callee-Save



Linker und Loader

- 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

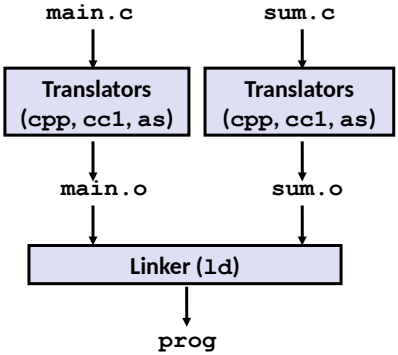
```
int sum(int *a, int n)
{
    int i, s = 0;

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

sum.c

Linker und Loader (cont.)

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



Linker und Loader (cont.)

\+ Modularität
- Programm in übersichtlichen kleinen Dateien
- Funktionen können wiederverwendet werden

⇒ vorgefertigte Programmbibliotheken

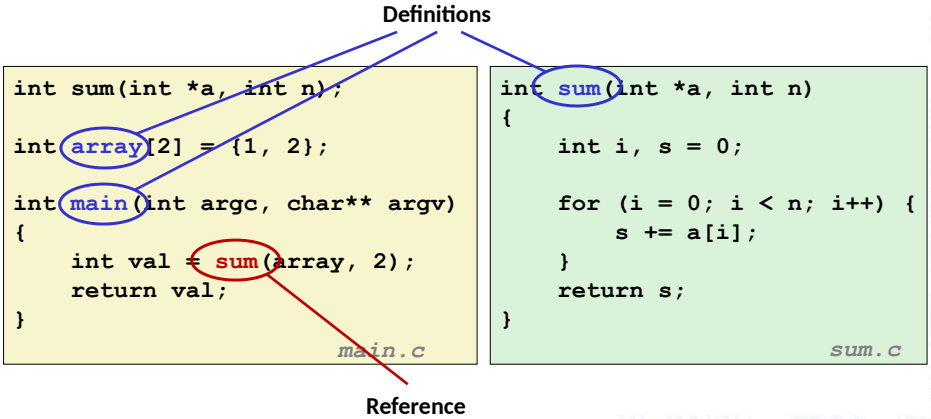
\+ Effizienz

⇒ 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

Aufgaben des Linkers

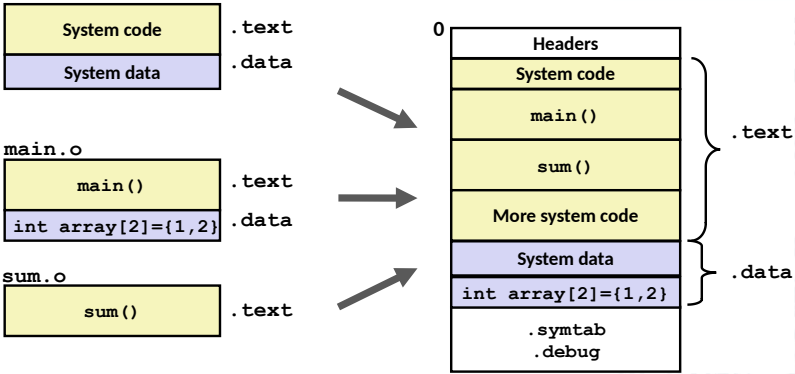
1. Symbole identifizieren (globale Variablen, Funktionen)
   Symbole auflösen (eindeutig machen) $\Rightarrow$ Symboltabelle



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*

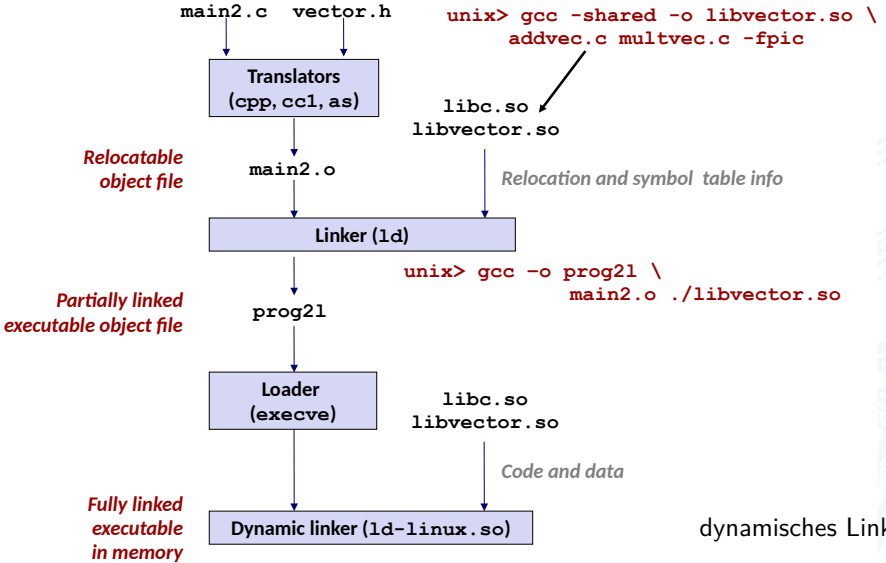
Statisches / dynamisches Linken

- 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

Assembler-Programmierung (cont.)

- *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"

Literatur

[BO15] R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective.
3rd global ed., Pearson Education Ltd., 2015. ISBN 978–1–292–10176–7
`csapp.cs.cmu.edu`

[TA14] A.S. Tanenbaum, T. Austin:
Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.
6. Auflage, Pearson Deutschland GmbH, 2014. ISBN 978–3–86894–238–5

[IA64] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture.*
Intel Corp.; Santa Clara, CA.
`software.intel.com/en-us/articles/intel-sdm`

Literatur (cont.)

[PH22] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition*.
6. Auflage, De Gruyter Oldenbourg, 2022. ISBN 978–3–11–075598–5

[PH20] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition*.
2nd edition, Morgan Kaufmann Publishers Inc., 2020. ISBN 978–0–12–820331–6

[Hyd10] R. Hyde: *The Art of Assembly Language*.
2nd edition, No Starch Press, 2010. ISBN 978–1–5932–7207–4
`www.randallhyde.com/AssemblyLanguage/www.artofasm.com`

[Hyd21] R. Hyde: *The Art of 64-bit Assembly, Volume 1*.
No Starch Press, 2021. ISBN 978–1–7185–0108–9
`artofasm.randallhyde.com`