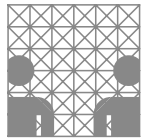


64-040 Modul InfB-RSB
Rechnerstrukturen und Betriebssysteme

https://tams.informatik.uni-hamburg.de/lectures/2021ws/vorlesung/rsb

– Kapitel 13 –

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

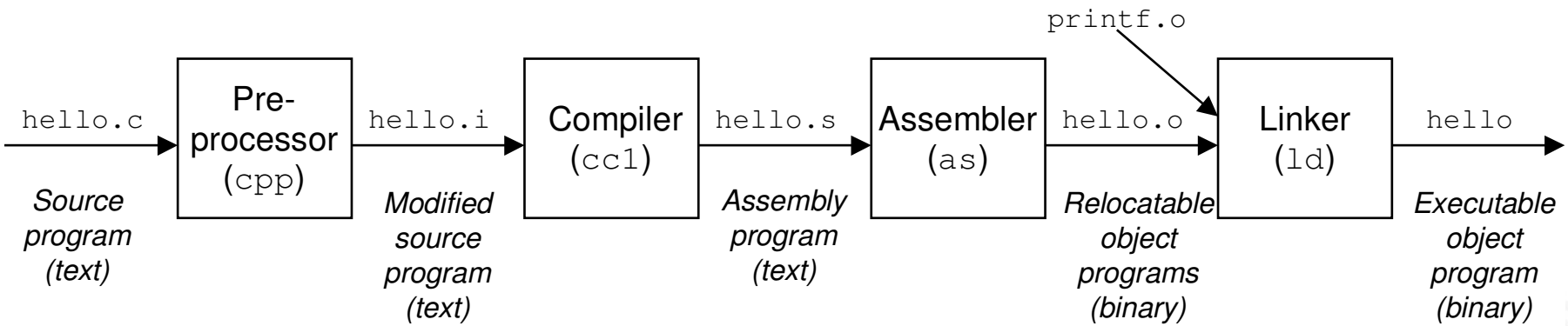
Wintersemester 2021/2022

Kapitel 13

Assembler-Programmierung

- Motivation
- Grundlagen der Assemblerebene
- x86 Assembler
  - Elementare Befehle + Adressierung
  - Operationen
  - Kontrollfluss
  - Sprungbefehle und Schleifen
  - Mehrfachverzweigung (Switch)
  - Funktionsaufrufe und Stack
- Speicherverwaltung
  - Elementare Datentypen
  - Arrays
  - Strukturen
- Linker und Loader
- Literatur

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

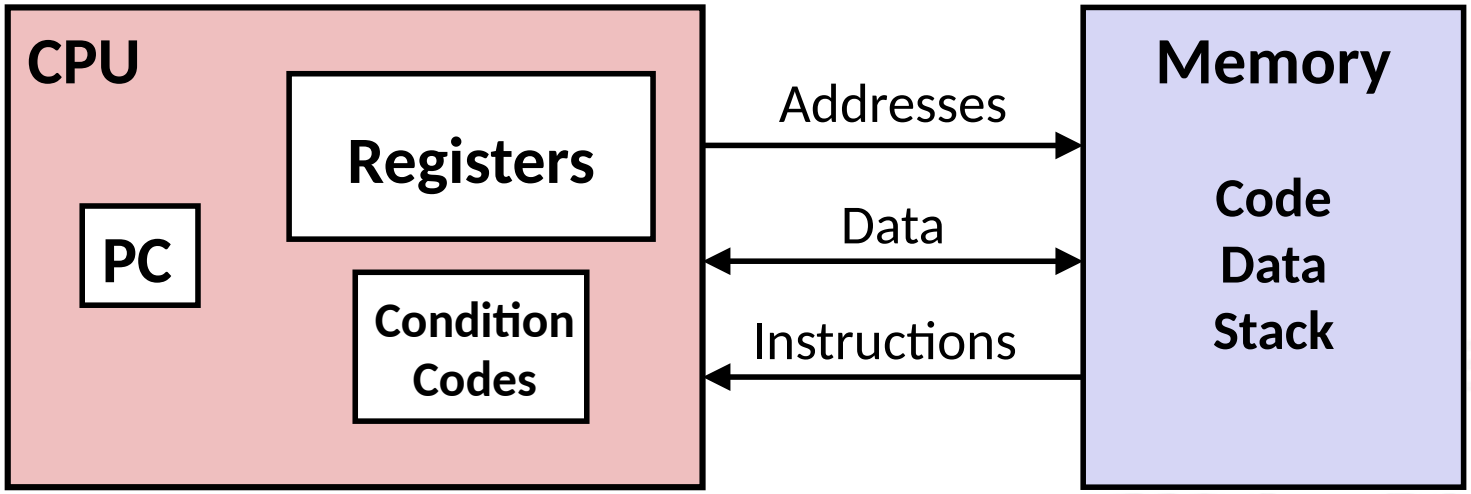
- Verständnis des Ausführungsmodells auf der Maschinenebene
- Programmverhalten bei Fehlern / Debugging
  - das High-Level Sprachmodell ist dort nicht anwendbar
- Programmleistung verstärken
  - Ursachen für Programm-Ineffizienz verstehen
  - effiziente „maschinengerechte“ Datenstrukturen / Algorithmen
- 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
  - Makros
  - Implementierung eines Assemblers (2-pass)
  - Tipps für effizientes Programmieren
  - Befehle für die Systemprogrammierung (supervisor mode)
  - x86 Gleitkommabefehle
  - ...

Assemblersicht des Programmierers



[BO15]

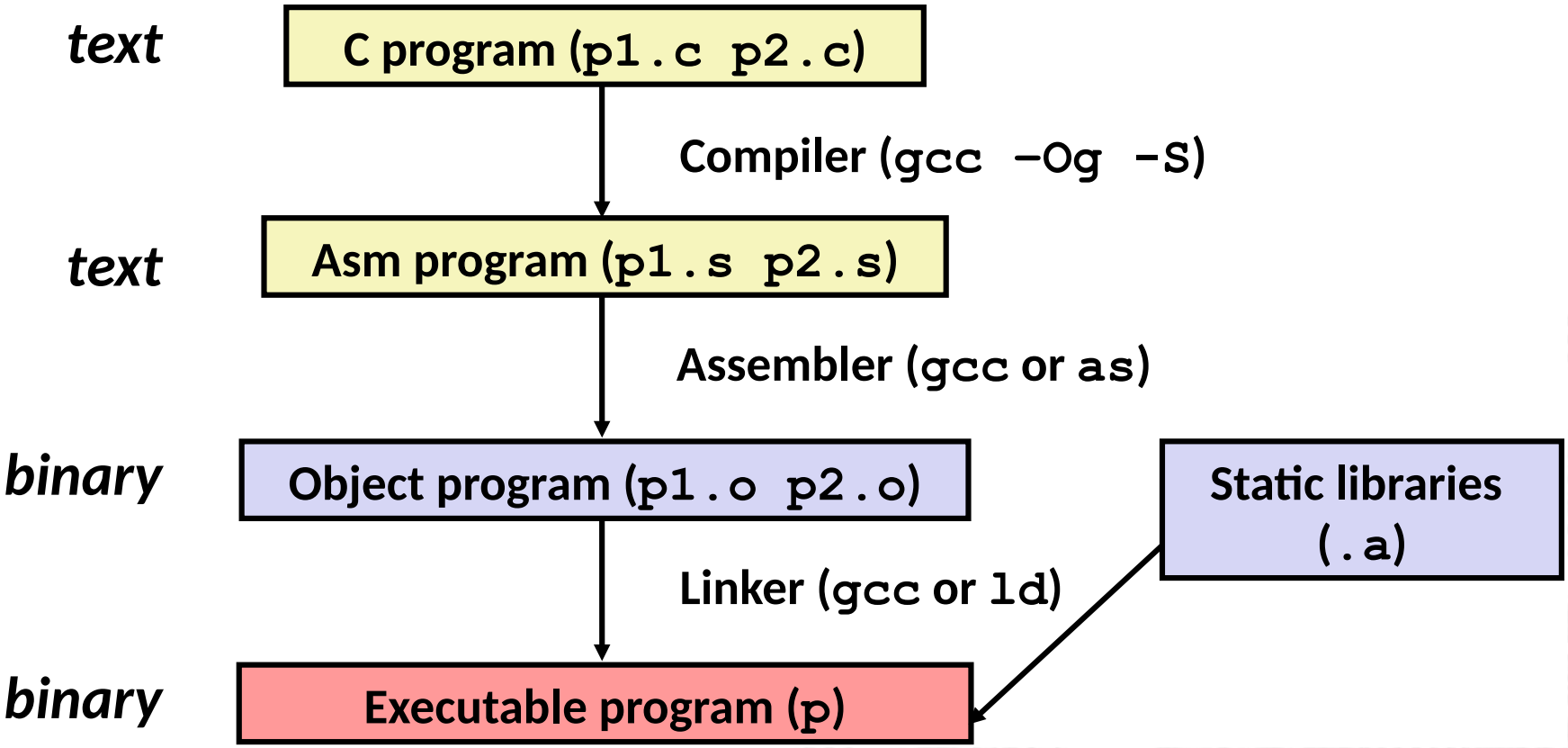
beobachtbare Zustände

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

Assemblersicht des Programmierers (cont.)

- Speicher
  - byteweise adressierbares Array
  - Code, Nutzerdaten, (einige) OS Daten
  - beinhaltet Kellerspeicher für Unterprogrammaufrufe „Stack“
  - –"– dynamischen Adressraum „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 Adressverweise

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

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


```
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
```

[BO15]

- `objdump -d ...`
  - Werkzeug zur Untersuchung des Objektcodes
  - rekonstruiert aus Binärcode den Assemblercode
  - kann auf vollständigem, ausführbaren Programm (`a.out`) 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 wie möglich)

x86 Assemblerprogrammierung

- Adressierungsarten
- arithmetische Operationen
- Statusregister
- Umsetzung von Programmstrukturen

Einschränkungen

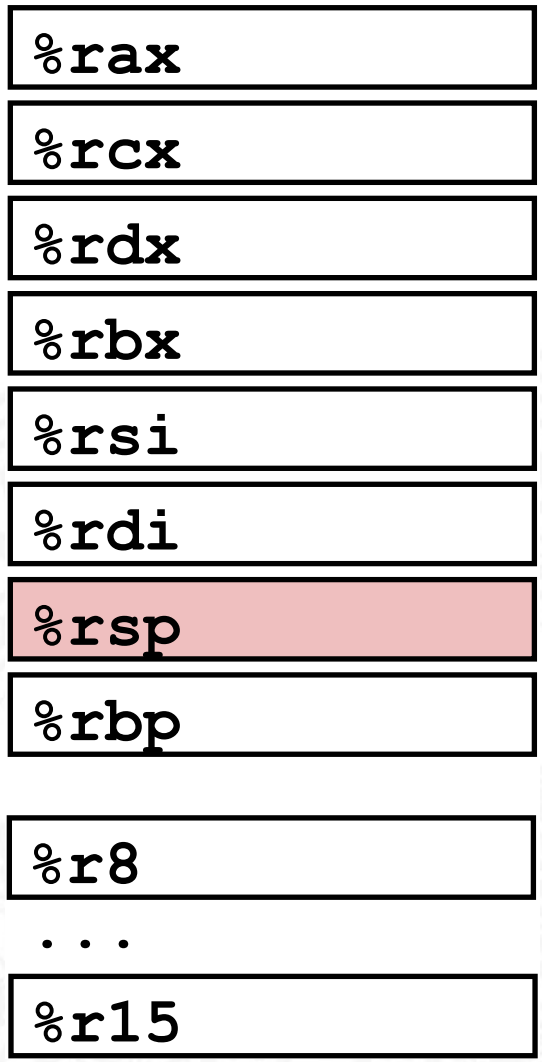
- Beispiele nutzen nur die 64-bit Datentypen
  `long` bei Linux (unter Windows nur 4-Byte!)
  - x86-64 wird wie 16-Register 64-bit Maschine benutzt (=RISC)
  - CISC Komplexität und Tricks bewusst vermieden
- Beispiele nutzen gcc/as Syntax (vs. Microsoft, Intel)

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

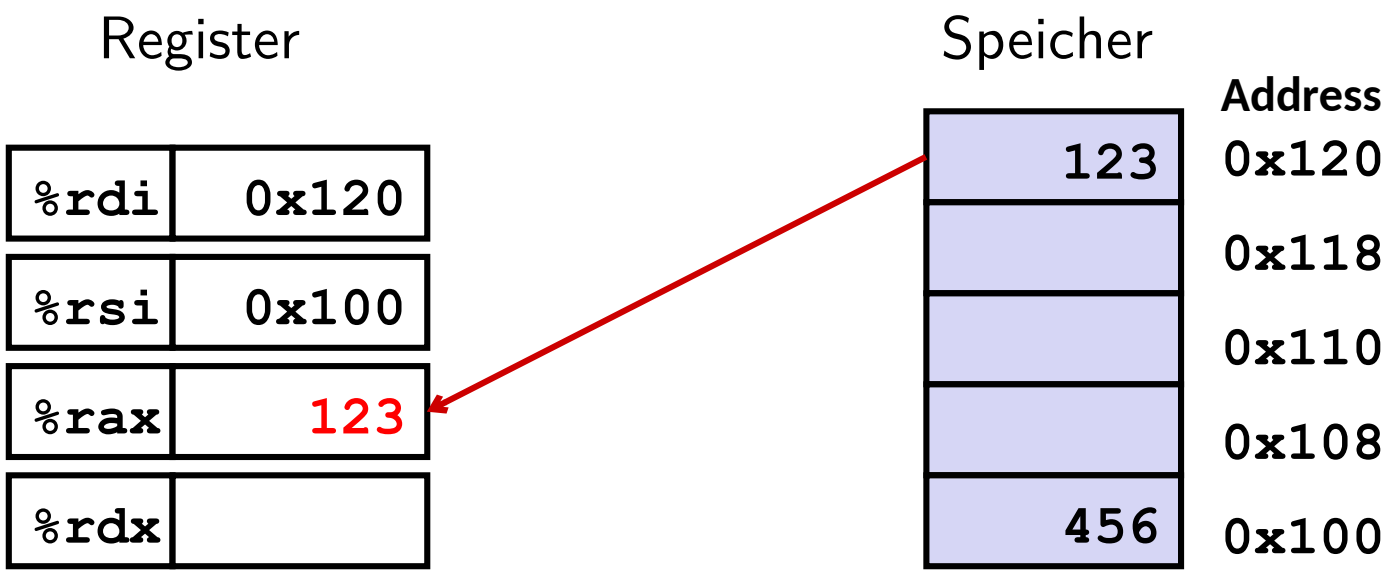
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Speicher

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

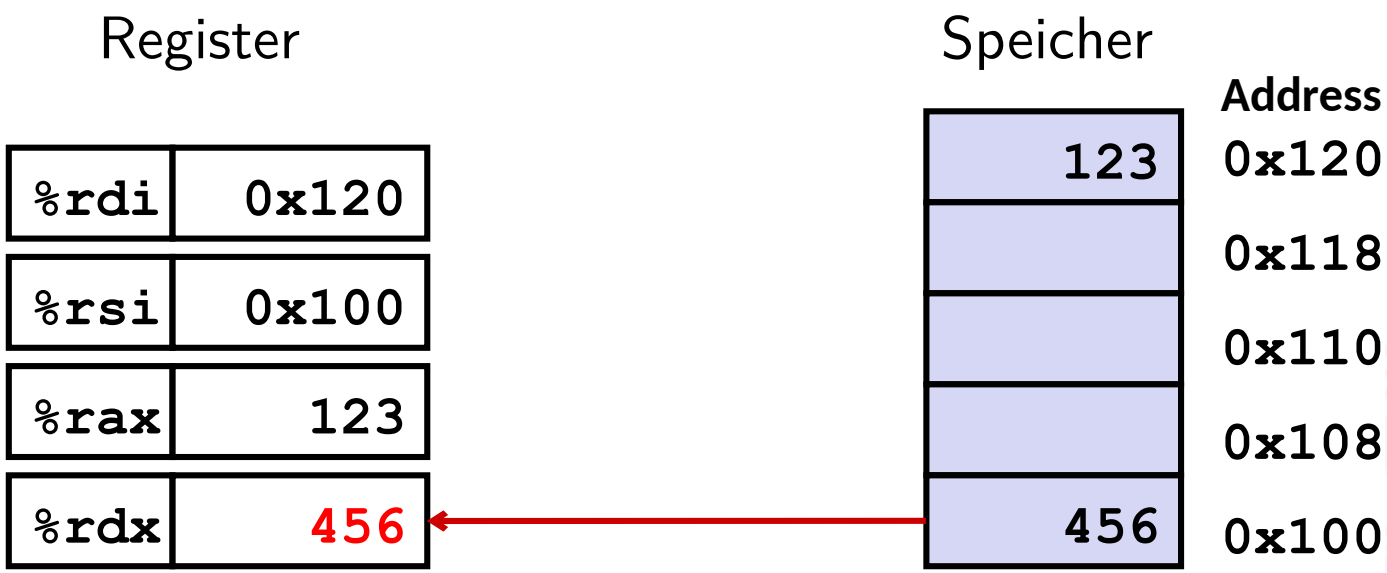
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Funktionsweise von swap()



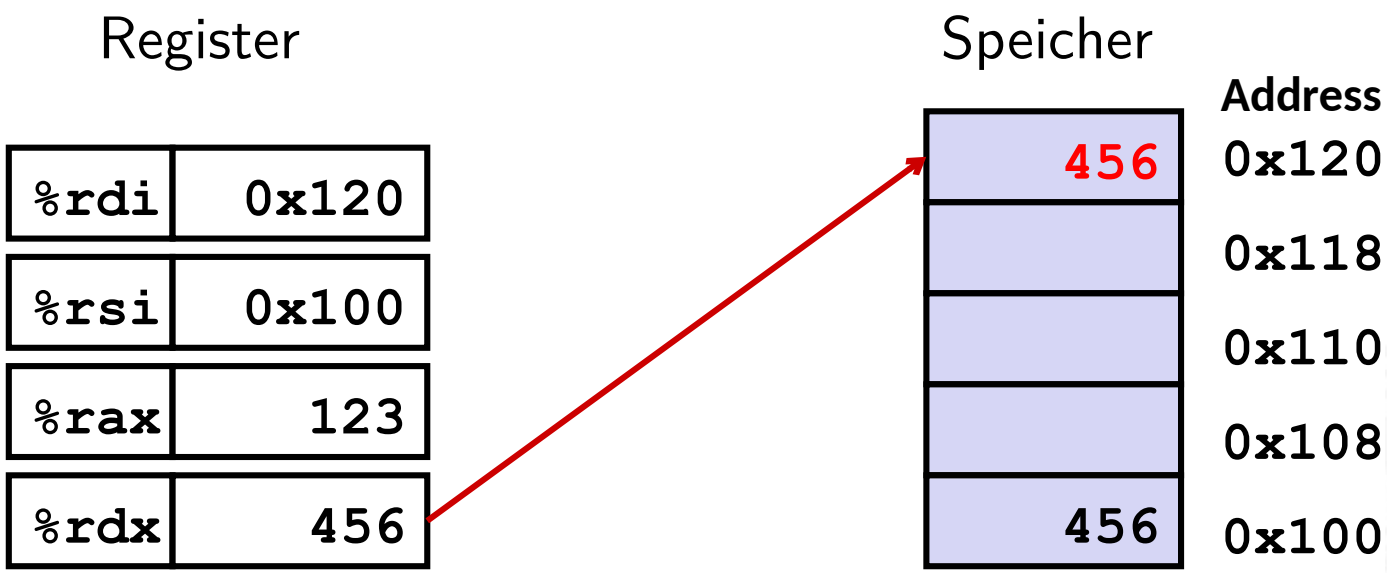

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Funktionsweise von swap()



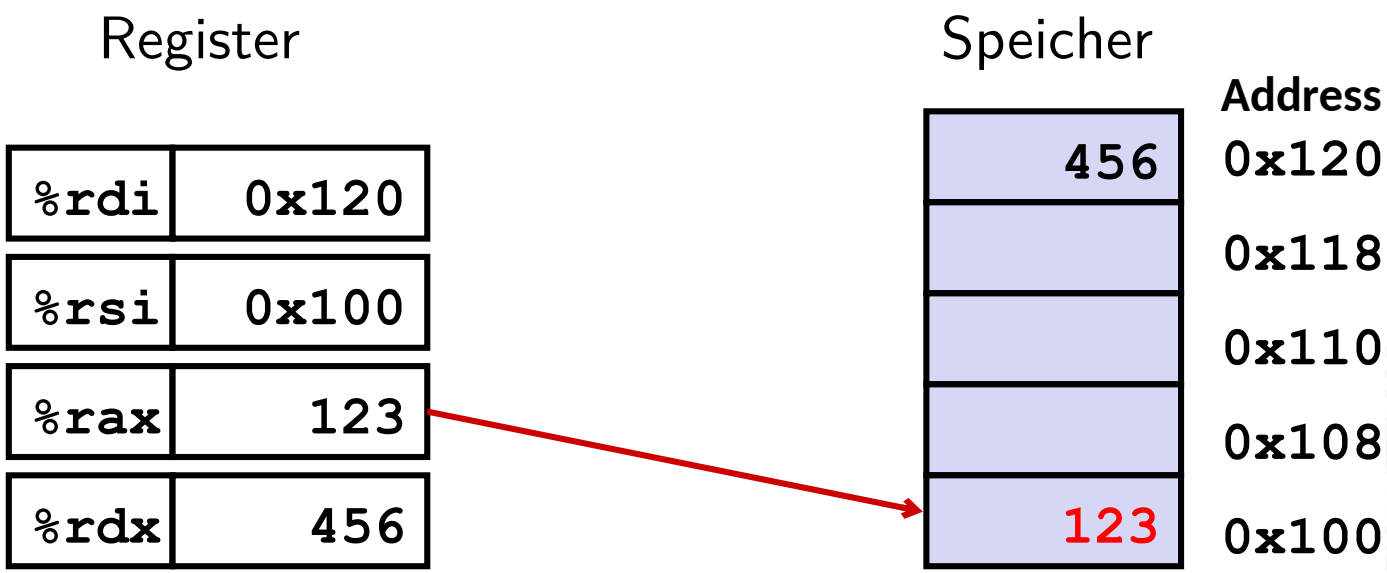

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

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;
}
```

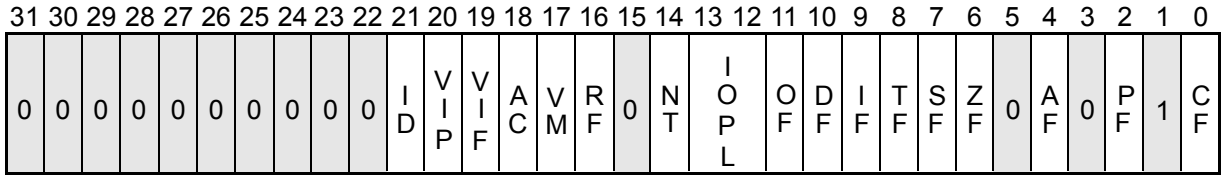
```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

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



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

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

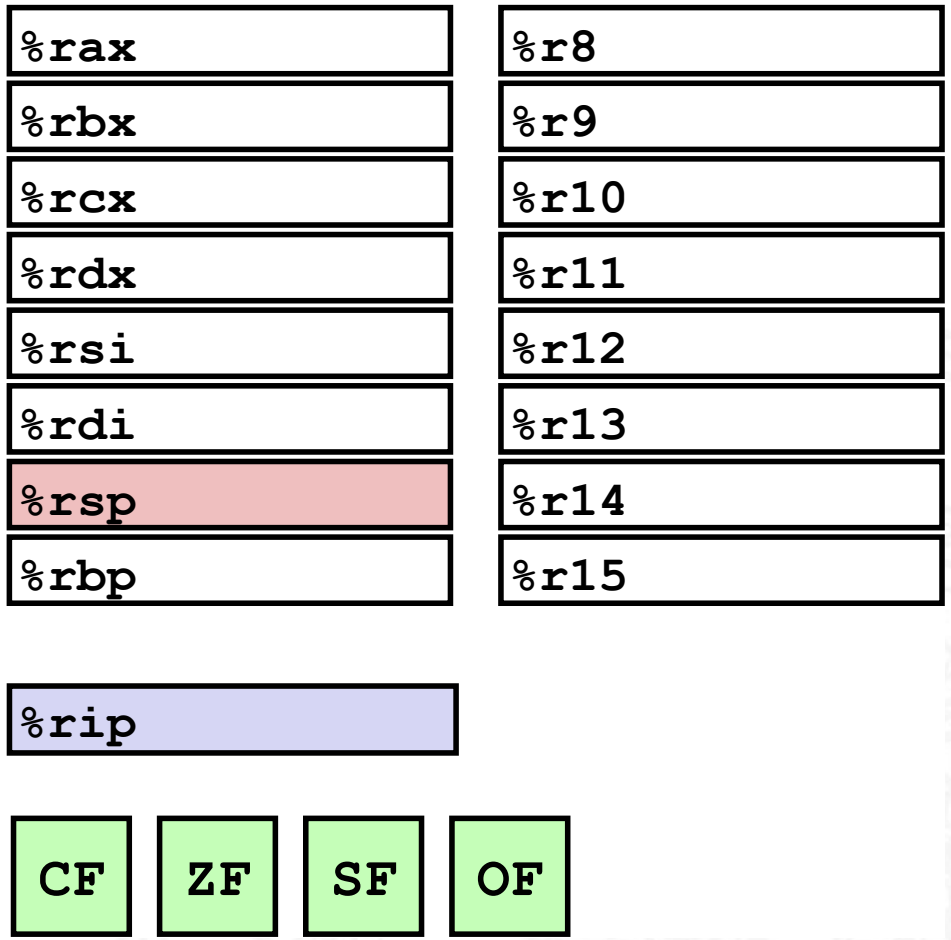
Reserved bit positions. DO NOT USE.
Always set to values previously read.

[IA64]

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

Prozessor aus Sicht des Programmierers

- temporäre Daten
  %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

- 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`
- kein Sprung

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

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

```
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
```

`if` Übersetzung – *conditional move* (cont.)

- \+ keine Sprünge (gut für Pipelining)
- – *beide Ausdrücke werden berechnet*
  - – Performanz, wenn komplizierte Berechnung
  - – Unsicher
  - – Seiteneffekte!
- ▶ Codeäquivalent

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

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

do ... while Übersetzung

- C Code

```
do
   Body
   while (Test);
```

goto-Version

```
loop:
   Body
   if (Test)
      goto loop
```

- beliebige Folge von C Anweisungen als Schleifenkörper
- Abbruchbedingung ist zurückgelieferter Integer Wert
  - $= 0$ entspricht Falsch: Schleife verlassen
  - $\neq 0$ –"– Wahr: nächste Iteration
- Rückwärtssprung setzt Schleife fort

do ... while Übersetzung (cont.)

- C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

do ... while Übersetzung (cont.)


```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

Register	Funktion
%rdi	Argument x
%rax	Rückgabewert

```
    movl    $0, %eax    #   result = 0
.L2:                    # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    #   t = x & 0x1
    addq    %rdx, %rax  #   result += t
    shrq    %rdi        #   x >>= 1
    jne     .L2         #   if(x) goto loop
    rep; ret
```

while Übersetzung – Sprung zu Test

- C Code

```
while (Test)
  Body
```

Sprung zu *Test* -Og

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

while Übersetzung – Sprung zu Test (cont.)

- C Code

```
long pcount_while
   (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Sprung zu *Test*

```
long pcount_goto_jtm
   (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

while Übersetzung – do ... while

- C Code -01

```
while (Test)
  Body
```

do ... while Äquivalent

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

goto-Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

while Übersetzung – do ... while (cont.)

- C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

do ... while

```
long pcount_goto_dw
   (unsigned long x) {
   long result = 0;
   if (!x) goto done;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
 done:
   return result;
}
```

for Übersetzung

- mehrstufige Übersetzung: `for ...` $\rightarrow$ `while` $\rightarrow$ ...

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

for Übersetzung (cont.)

- C Code

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  if (!(i < WSIZE))
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
  if (i < WSIZE)
    goto loop;
 done:
  return result;
}
```

Init

!Test

Body

Update

Test

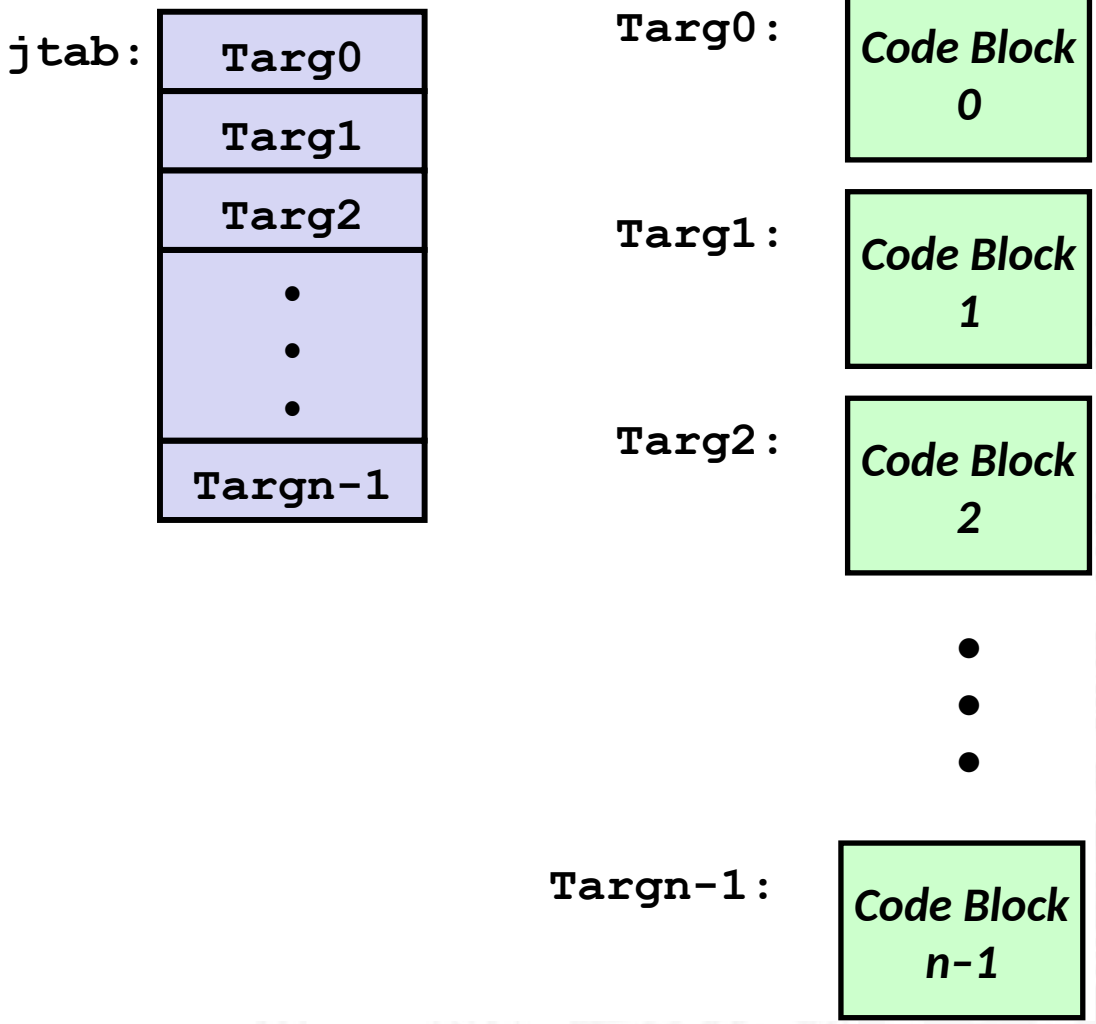
switch Mehrfachverzweigungen

- Implementierungsoptionen
  1. Folge von „If-then-else“
     - \+ gut bei wenigen Alternativen
     - – langsam bei vielen Fällen
  2. Sprungtabelle „Jump Table“
     - Vermeidet einzelne Abfragen
     - möglich falls Alternativen kleine ganzzahlige Konstanten sind
  - Compiler (gcc) wählt eine der beiden Varianten entsprechend der Fallstruktur

```
long my_switch
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Sprungtabelle


```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
     • • •
  case val_n-1:
    Block n-1
}
```




```
goto *JTab[x];
```

- Vorteil: k-fach Verzweigung in $\mathcal{O}(1)$ Operationen

Sprungtabelle (cont.)

C Code

```
long my_switch(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Sprungtabelle

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
my_switch:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # use default
    jmp     *.L4(,%rdi,8)   # goto *Jtab[x]
```

Register	Funktion
%rdi	Arg. x
%rsi	Arg. y
%rdx	Arg. z

Sprungtabelle (cont.)

- Compiler erzeugt Code für jeden `case` Zweig
  - je ein Label am Start der Zweige: `.L8`, `.L3`, `.L5`...
  - werden dann vom Assembler/Linker in Adressen umgesetzt
- Tabellenstruktur
  - jedes Ziel benötigt 4 Bytes
  - Basisadresse bei `.L4`
- Sprünge
  - Direkt: `jmp .L8`
  - Indirekt: `jmp *.L4(,%rdi,8)`
    - Start der Sprungtabelle: `.L4`
    - Register `%rdi` speichert x
    - Skalierungsfaktor 8 für Tabellenoffset
    - Sprungziel: effektive Adresse `.L4` $+ x \times 8$

Sprungtabelle (cont.)


```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

Mechanismen für Prozeduraufrufe

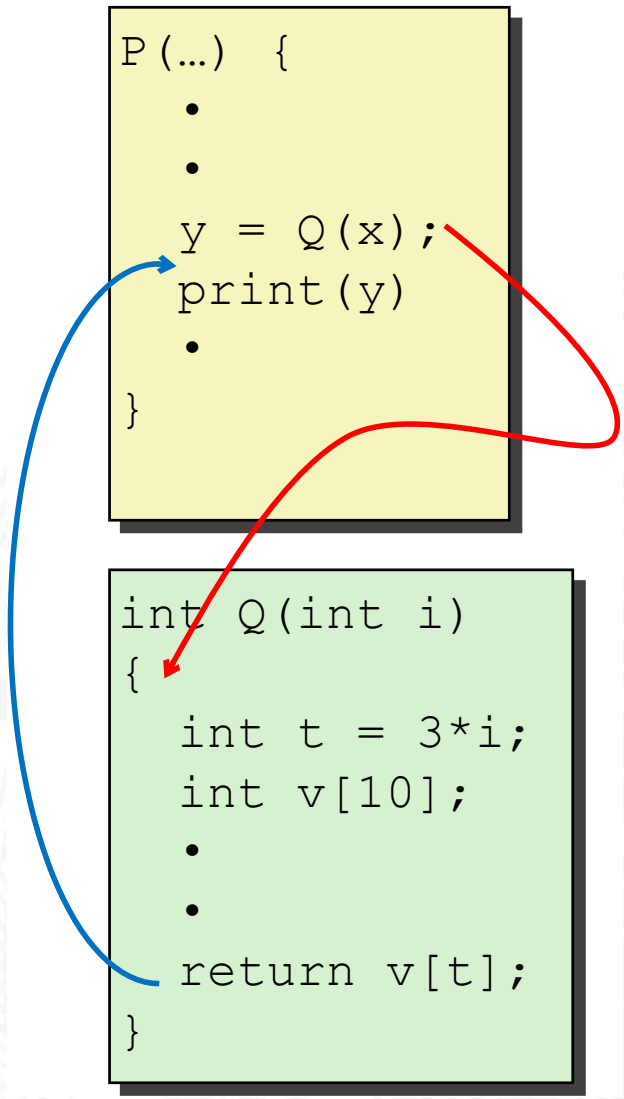
- Kontrollübergabe
  - zu Unterprogrammcode
  - zurück zu Aufruf
- Datenübergabe
  - Argumente
  - Rückgabewert
- Speicherverwaltung
  - Allokation während der Ausführung
  - Freigabe nach `return`

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

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

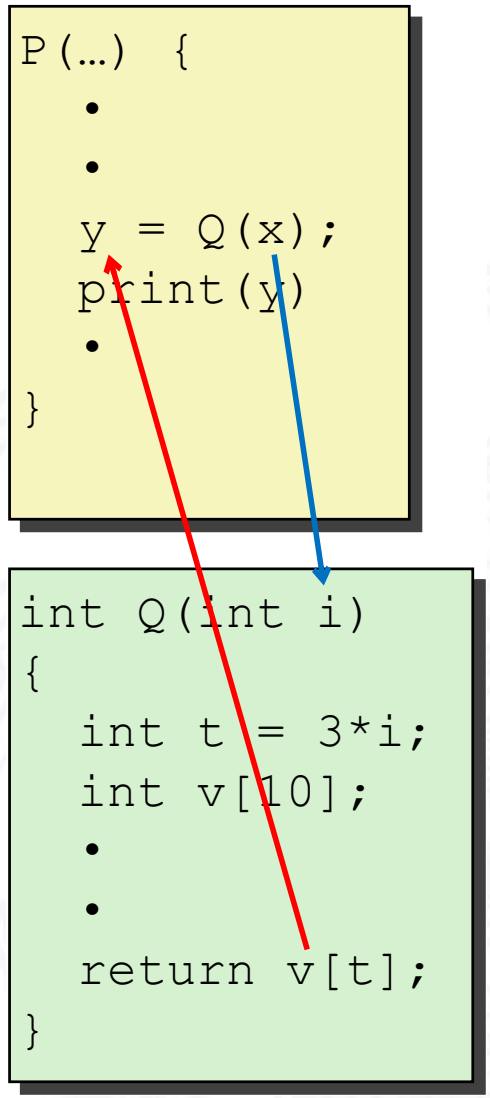
Mechanismen für Prozeduraufrufe

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



Mechanismen für Prozeduraufrufe

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



Mechanismen für Prozeduraufrufe

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

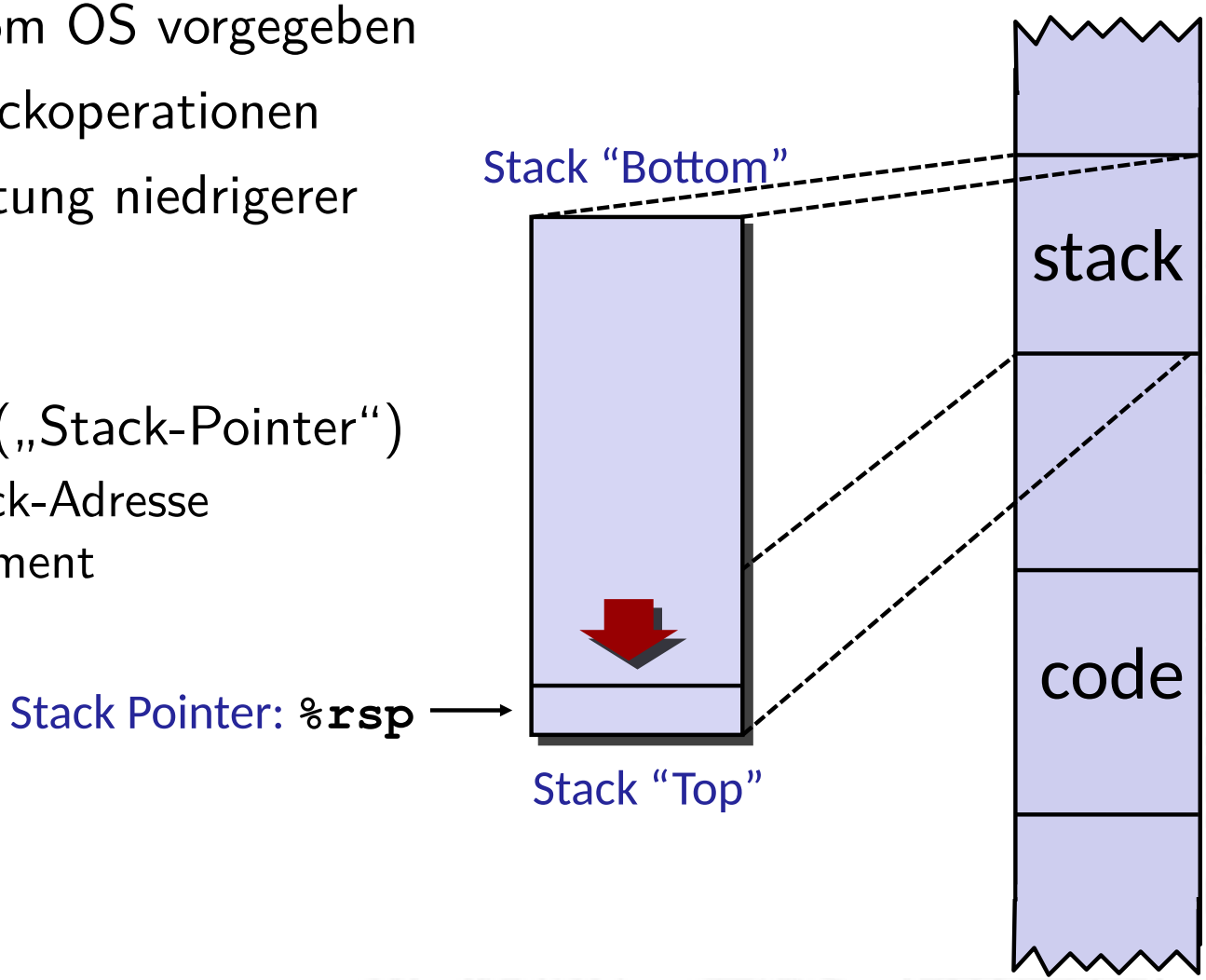
```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
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



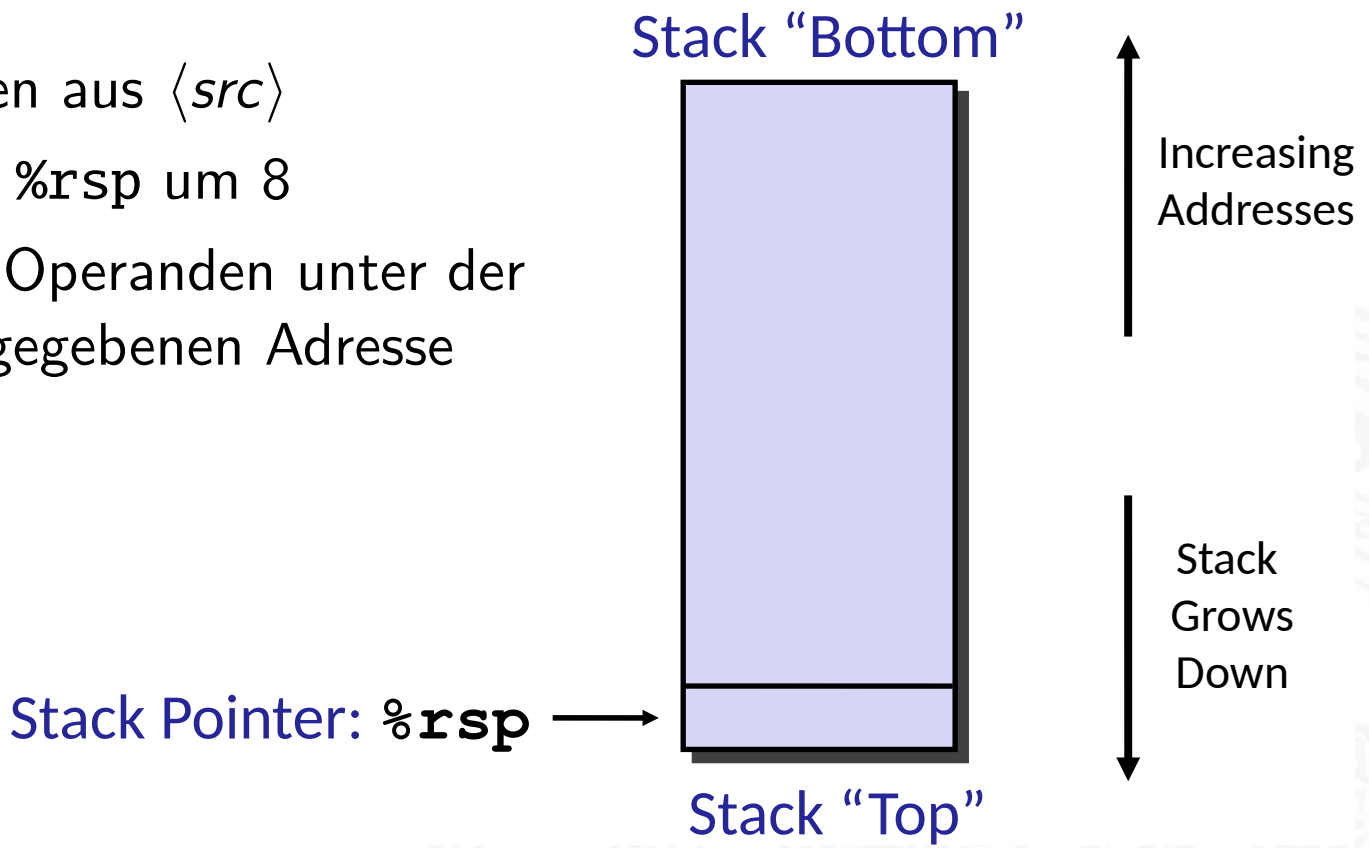
Stack (Kellerspeicher) (cont.)

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

Stack: Push

`pushq` ⟨*src*⟩

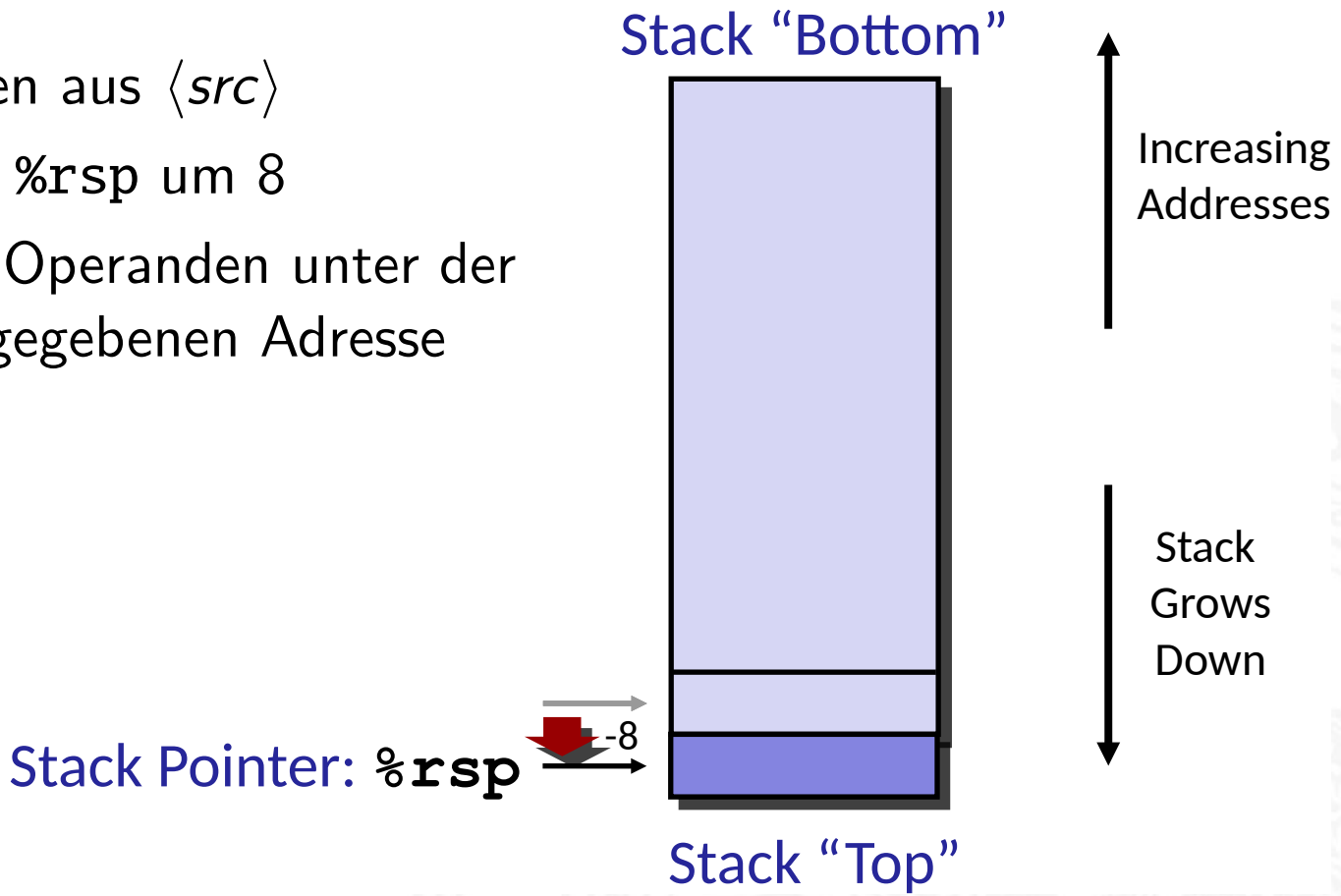
- holt Operanden aus ⟨*src*⟩
- dekrementiert `%rsp` um 8
- speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Push

`pushq` ⟨*src*⟩

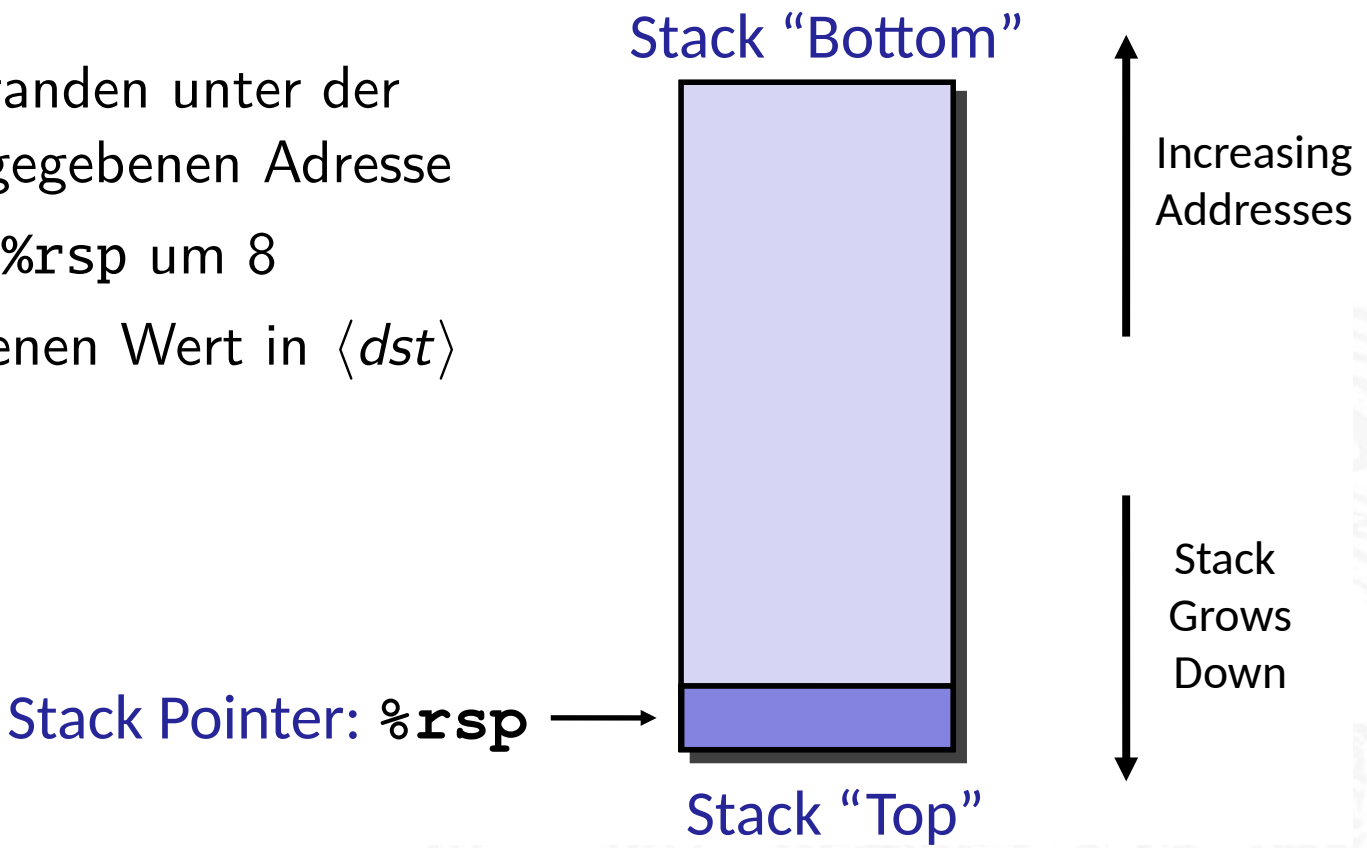
- holt Operanden aus ⟨*src*⟩
- dekrementiert `%rsp` um 8
- speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Pop

`popq` ⟨*dst*⟩

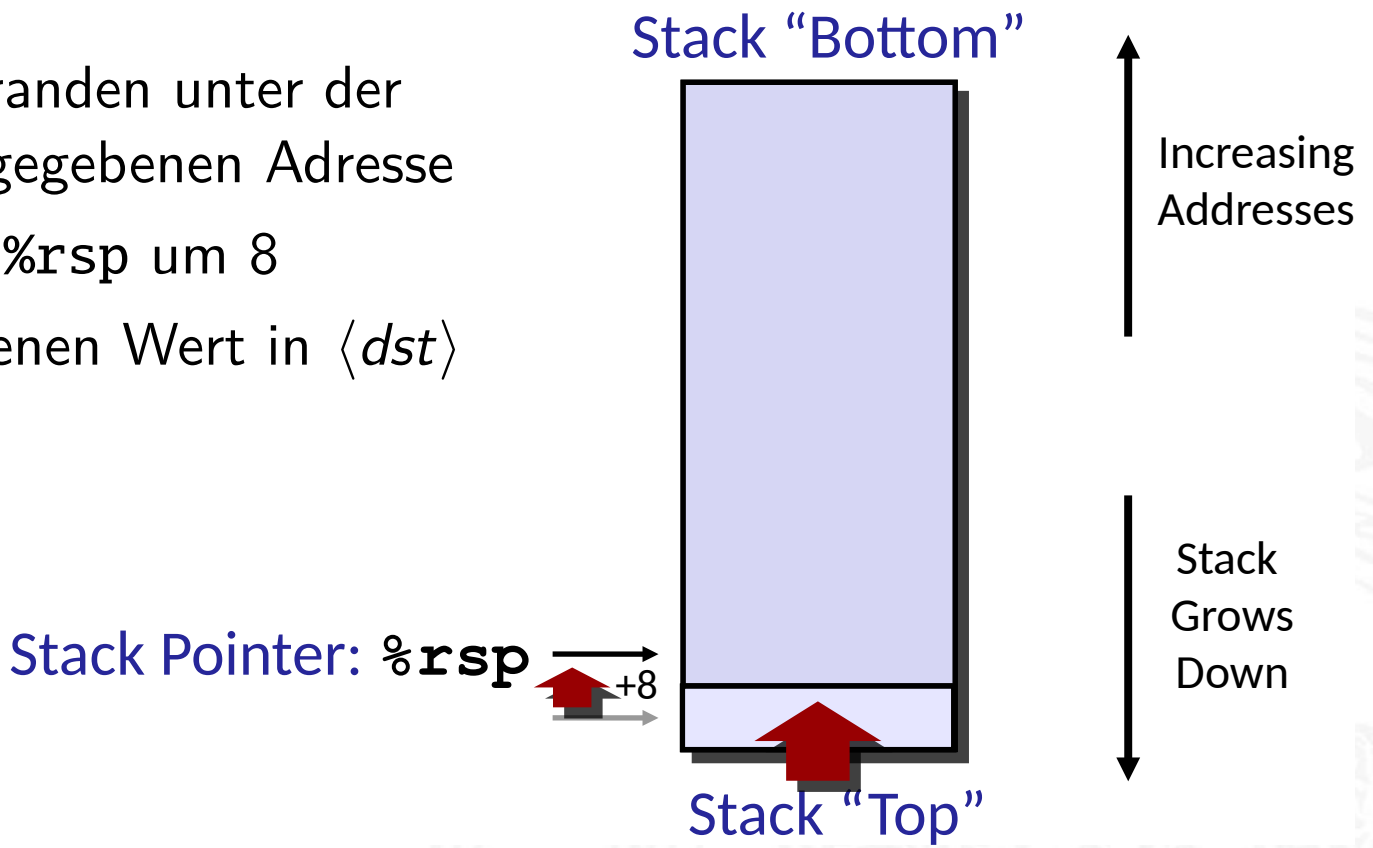
- liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- inkrementiert `%rsp` um 8
- schreibt gelesenen Wert in ⟨*dst*⟩



Stack: Pop

`popq` $\langle dst \rangle$

- liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- inkrementiert `%rsp` um 8
- schreibt gelesenen Wert in $\langle dst \rangle$



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
- Stack zur Unterstützung von `call` und `ret`
- Register mit Spezialaufgaben
  - `%rsp` „stack-pointer“: Speicheradresse des top-of-stack
  - `%rbp` „base-pointer“: Speicheradresse des aktuellen Frame (s.u.)
- Prozeduraufruf: `call` ⟨*label*⟩
  - Rücksprungadresse auf Stack („Push“)
  - Sprung zu ⟨*label*⟩
- Wert der Rücksprungadresse
  - Adresse der auf den `call` folgenden Anweisung
- Rücksprung `ret`
  - Rücksprungadresse vom Stack („Pop“)
  - Sprung zu dieser Adresse

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;
}
```

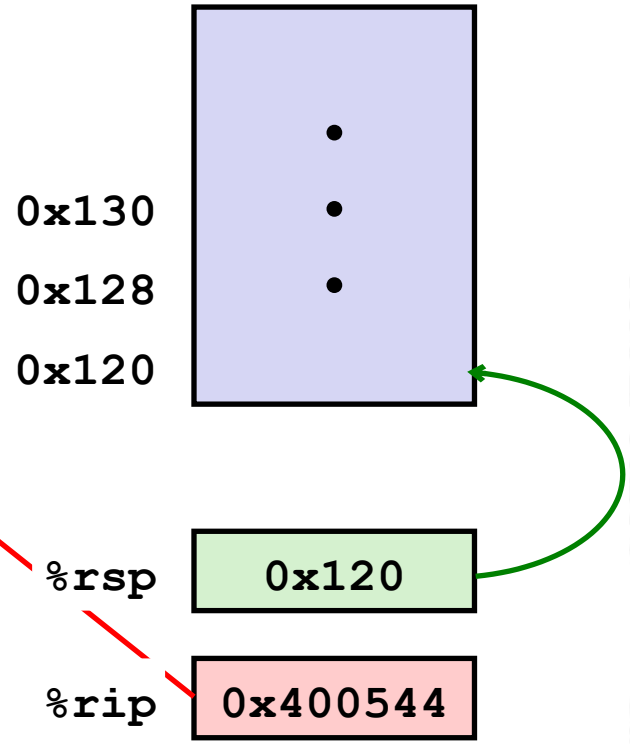
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

Kontrollübergabe

- Prozeduraufruf `callq`

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

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

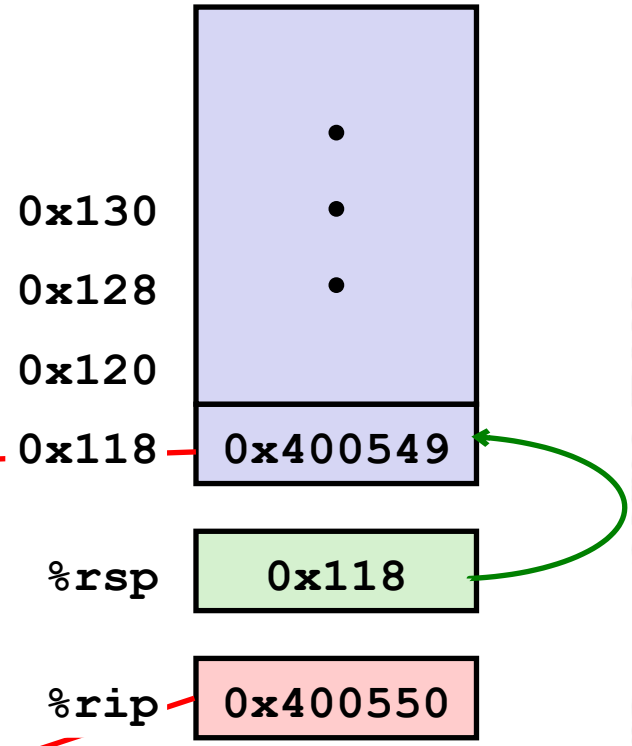


Kontrollübergabe (cont.)

- Rücksprungadresse auf Stack
- Programmzähler setzen `%rip`

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

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

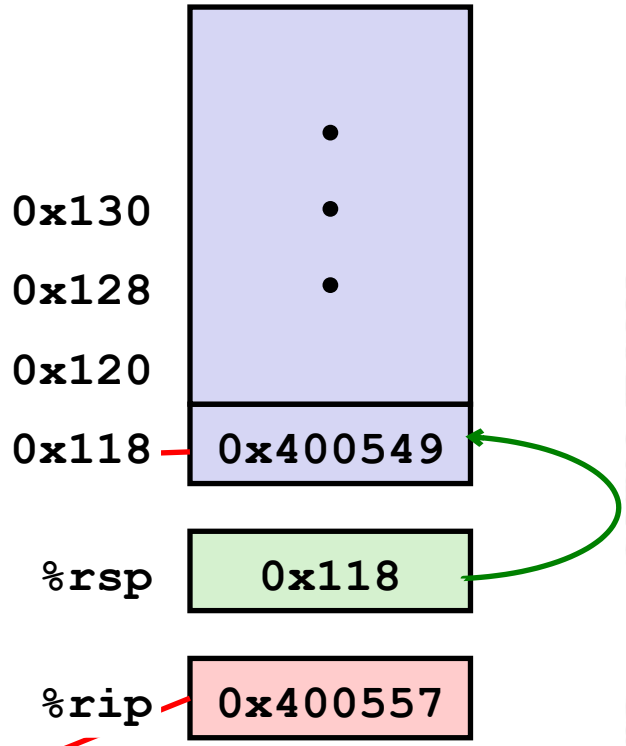


Kontrollübergabe (cont.)

- Rücksprung `retq`

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

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

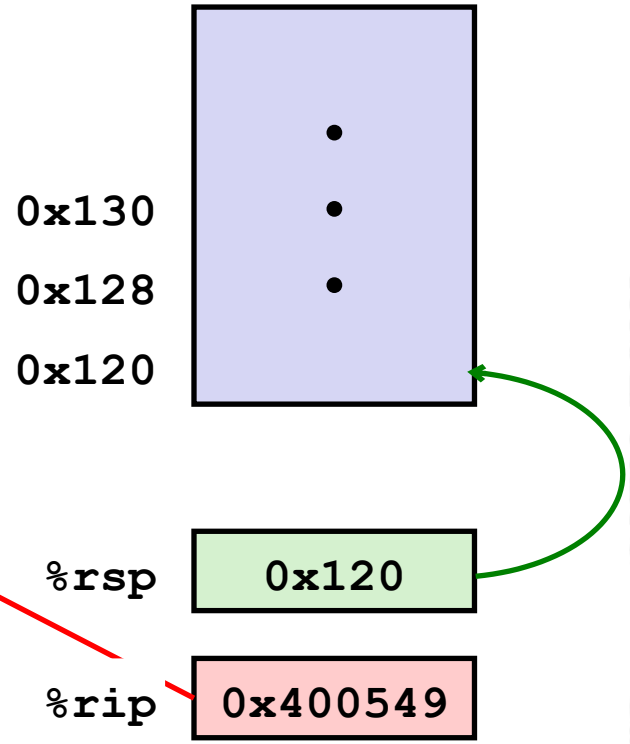


Kontrollübergabe (cont.)

- Rücksprungadresse vom Stack
- Programmzähler setzen `%rip`

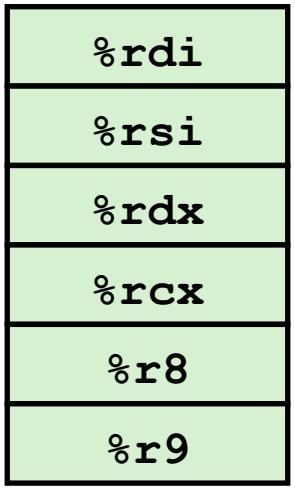
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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

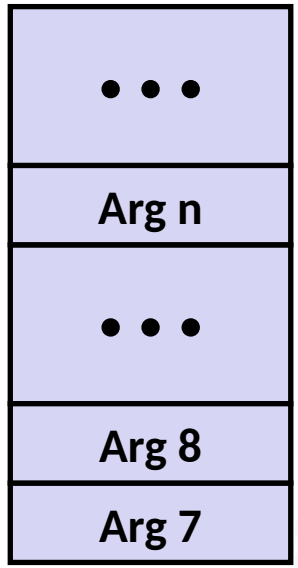


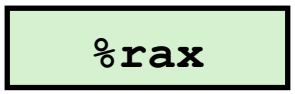
Datenübergabe

- 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;
}
```

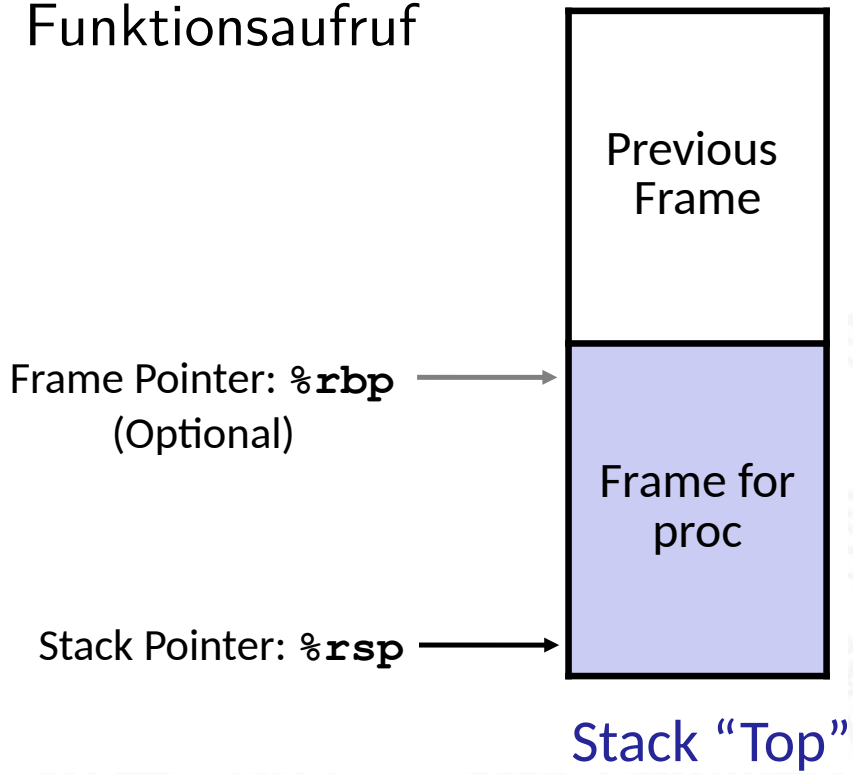
```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  # s in %rax
  400557:  retq               # Return
```

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 aufrufendem Programm („Caller“) beendet
- Stack-„Frame“
  - der Bereich/Zustand einer einzelnen Prozedur-Instanziierung

Stack-Frame

- „Closure“: alle Daten für einen Funktionsaufruf
- Daten
  - Rücksprungadresse
  - ggf. lokale Variablen
  - ggf. temporäre Daten



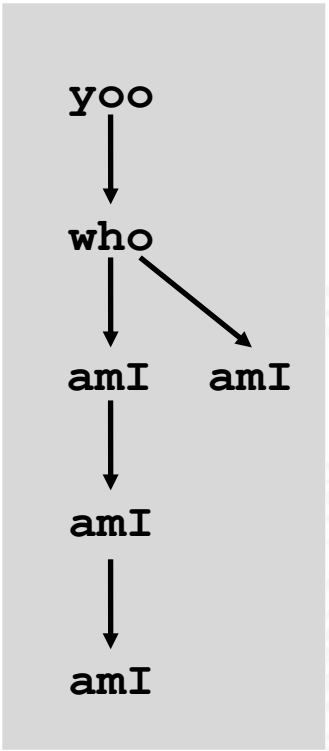
- Speicherverwaltung durch Funktion
  - bei Aufruf wird Stack gefüllt: „Set-up“ Code
  - bei Return wieder freigegeben: „Finish“ Code

Beispiel: Prozeduraufrufe

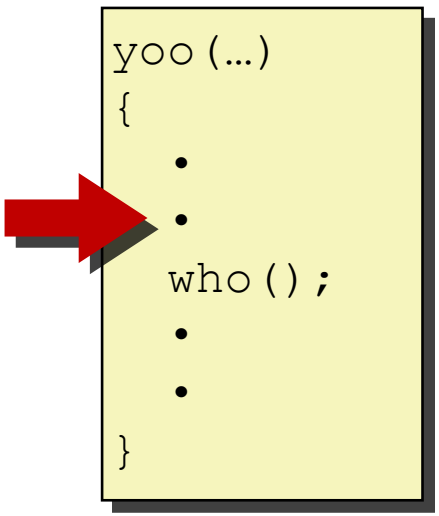

```
yoo(...)
{
   •
   •
   who();
   •
   •
}
```

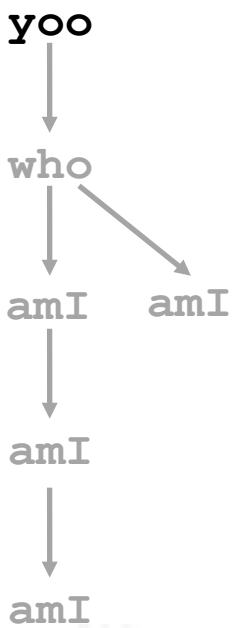
```
who(...)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

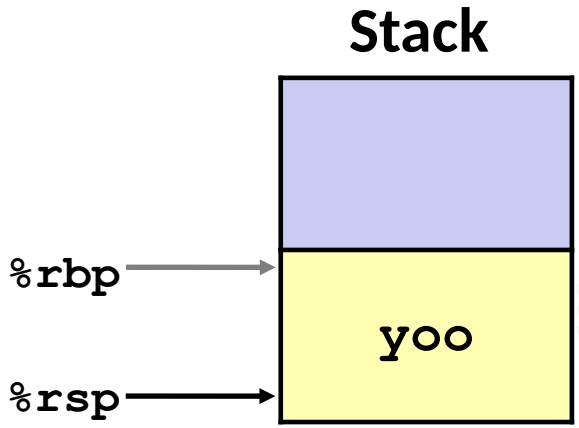
```
amI(...)
{
   •
   •
   amI();
   •
   •
}
```



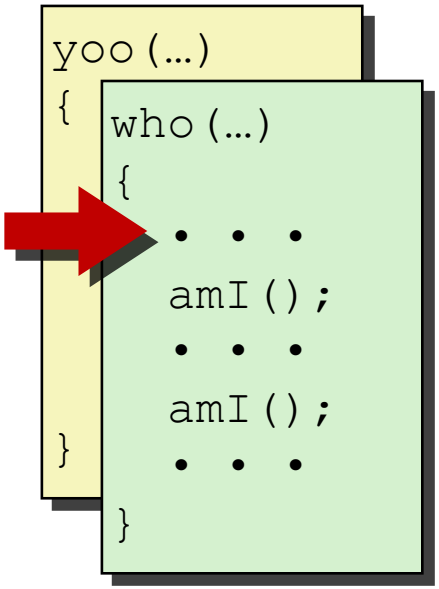
Beispiel: Prozeduraufrufe (cont.)

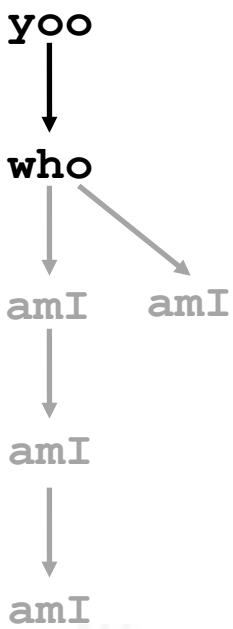


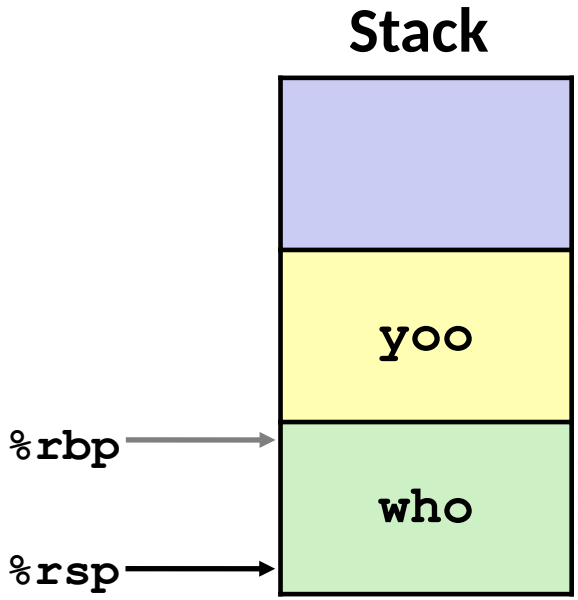




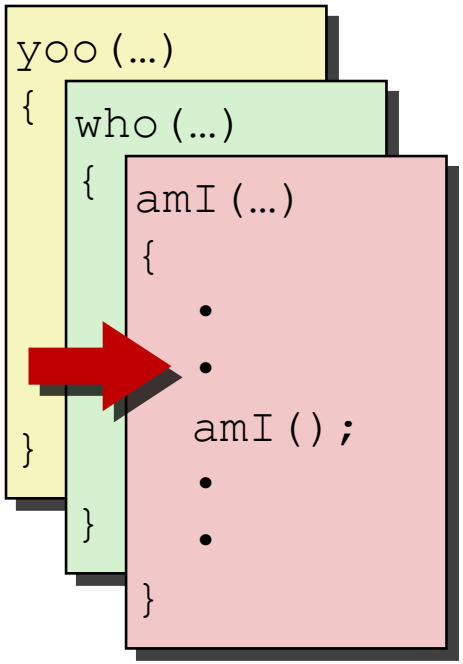
Beispiel: Prozeduraufrufe (cont.)

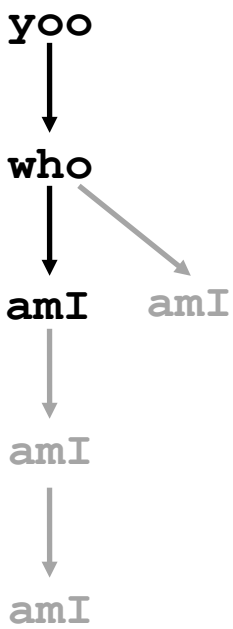


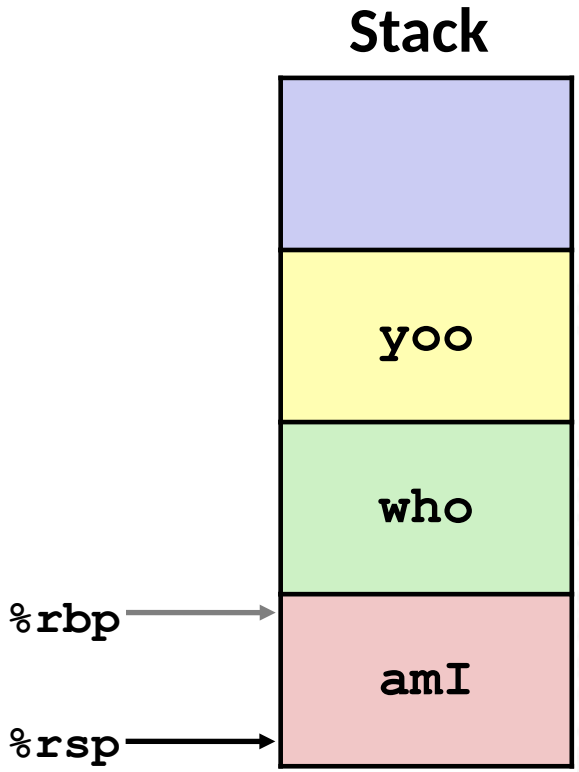




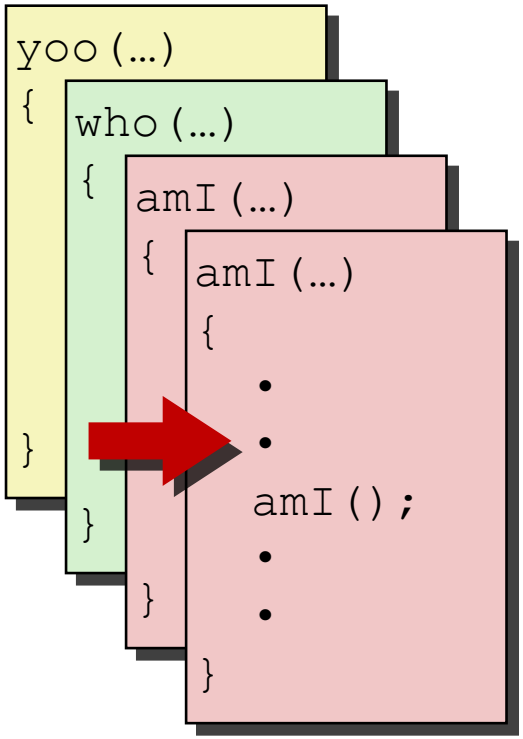
Beispiel: Prozeduraufrufe (cont.)

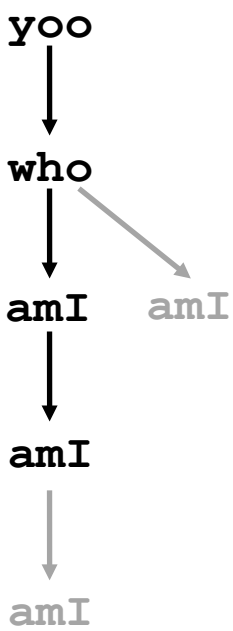


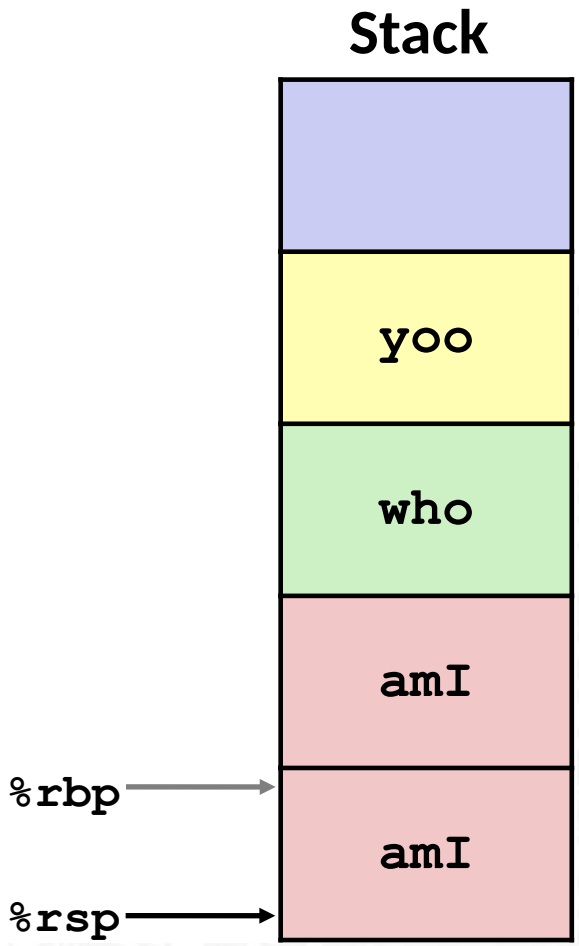




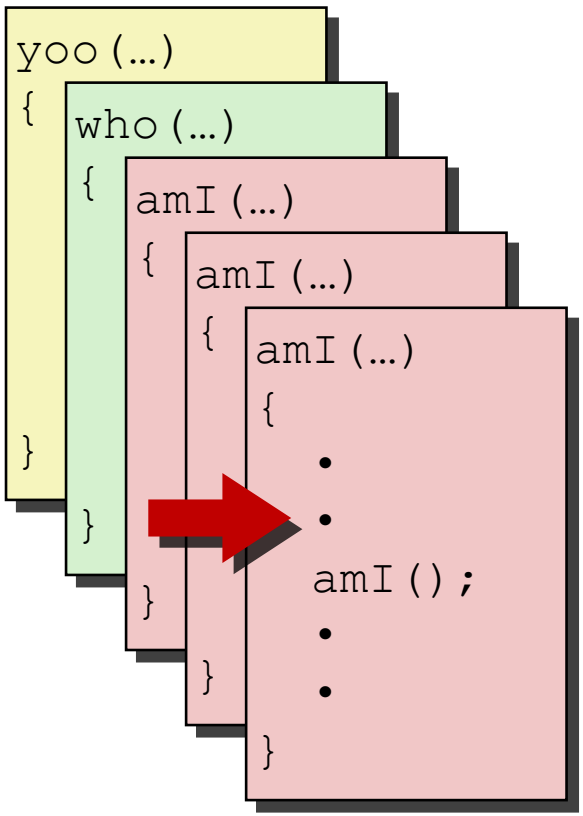
Beispiel: Prozeduraufrufe (cont.)

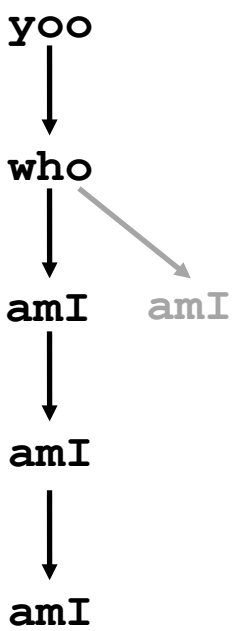


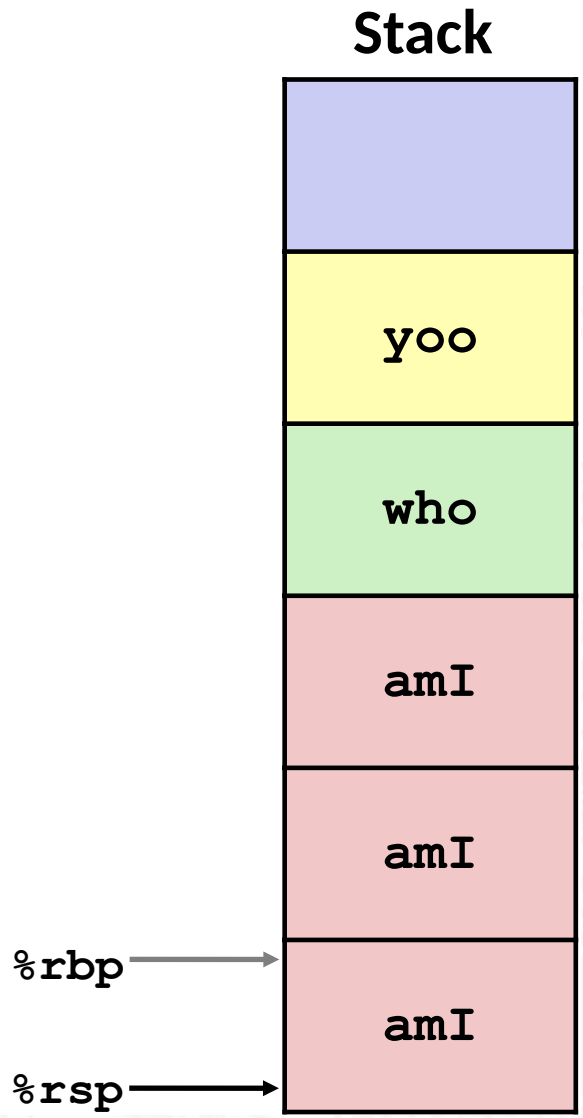




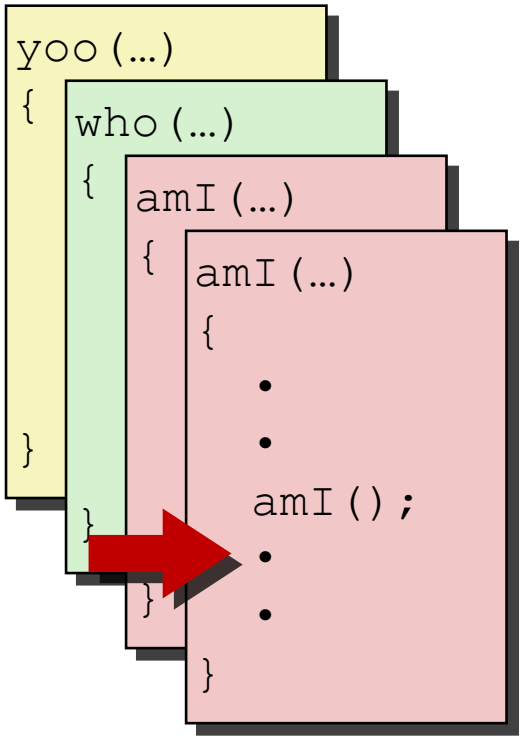
Beispiel: Prozeduraufrufe (cont.)

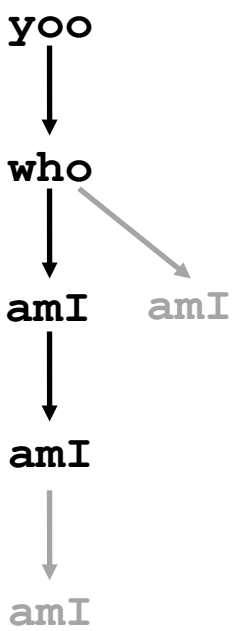


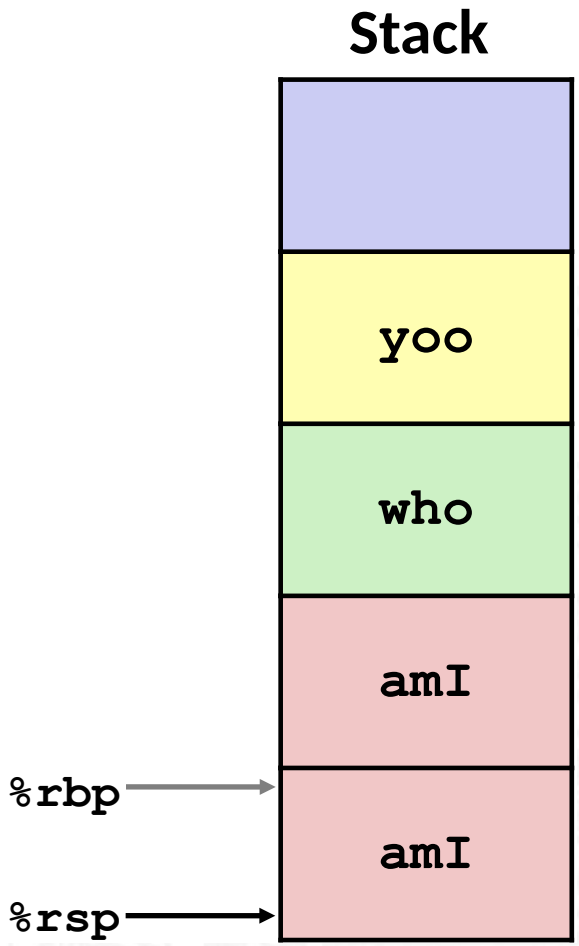




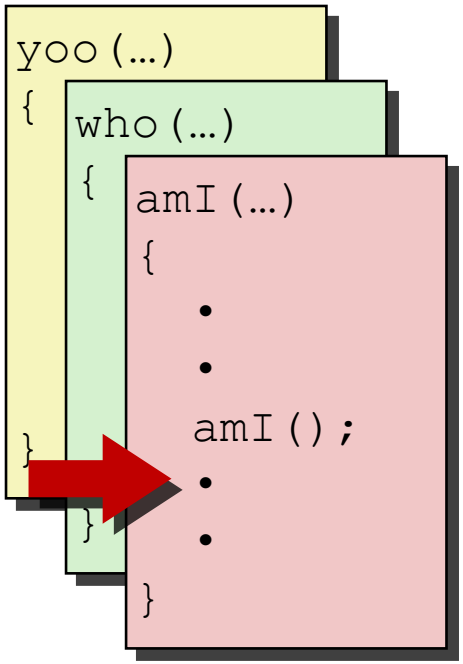
Beispiel: Prozeduraufrufe (cont.)

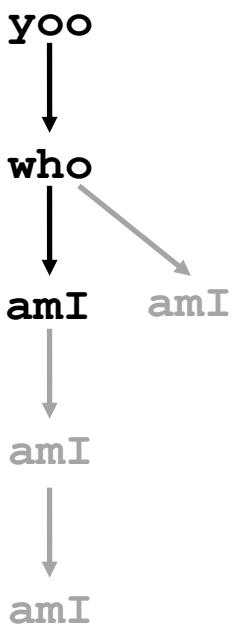


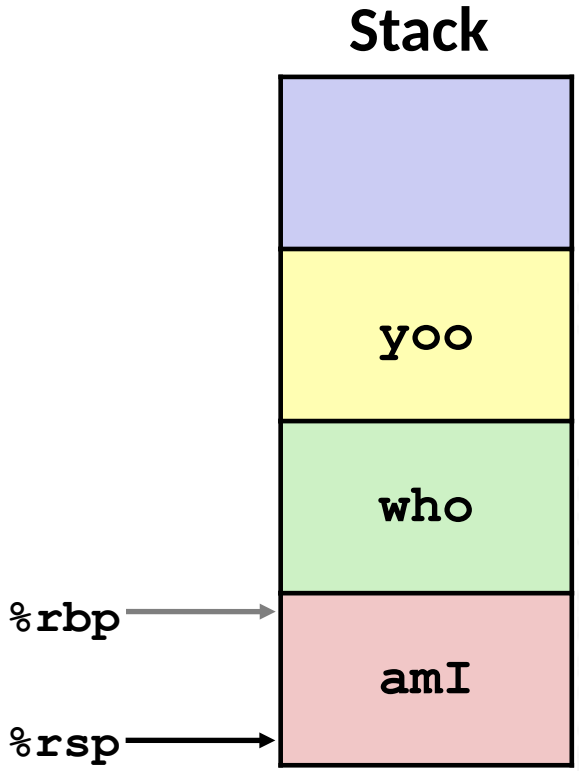




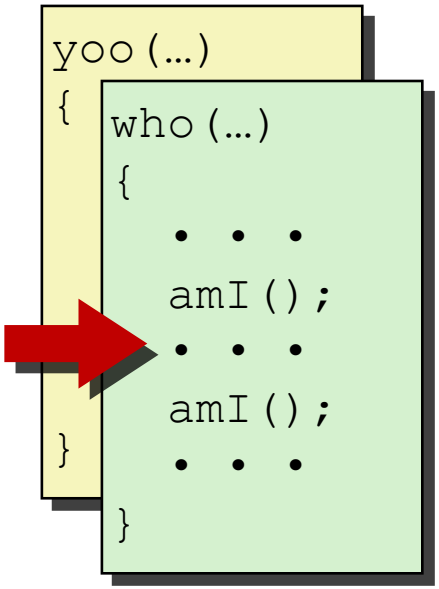
Beispiel: Prozeduraufrufe (cont.)

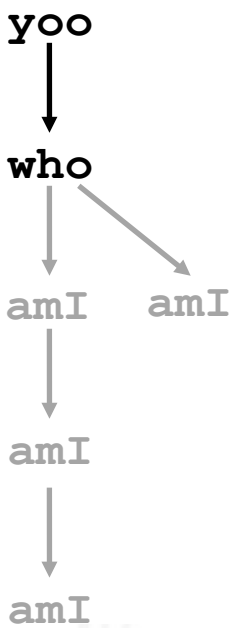


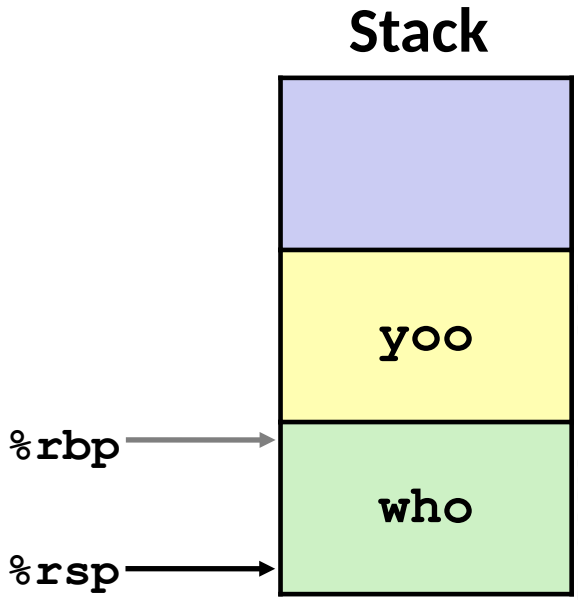




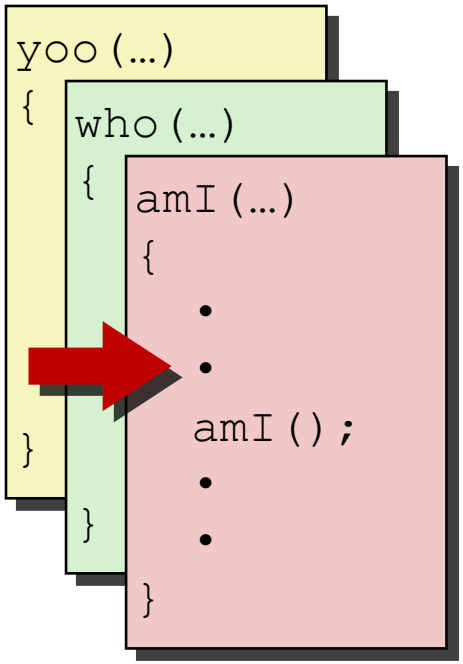
Beispiel: Prozeduraufrufe (cont.)

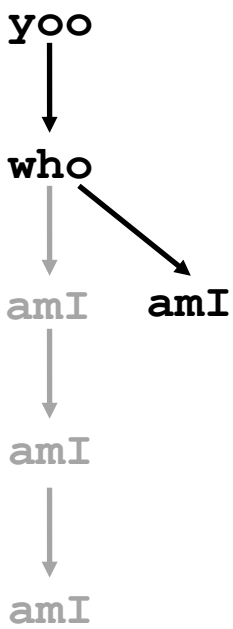


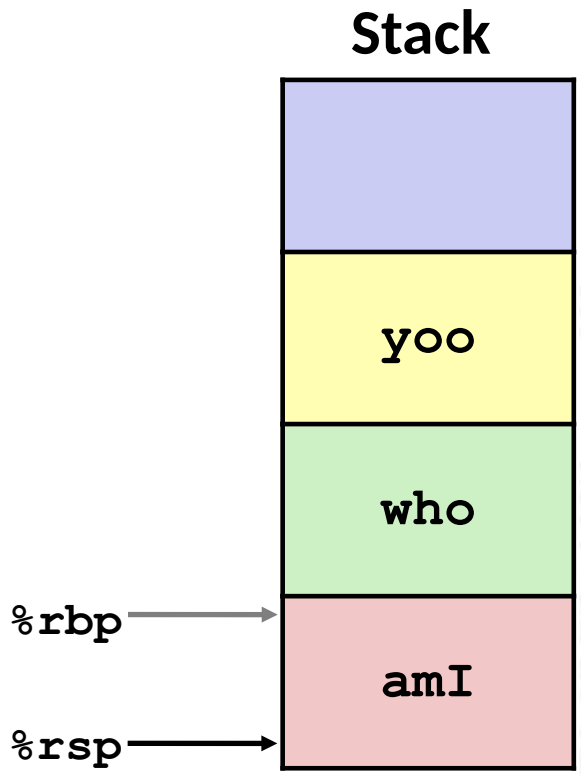




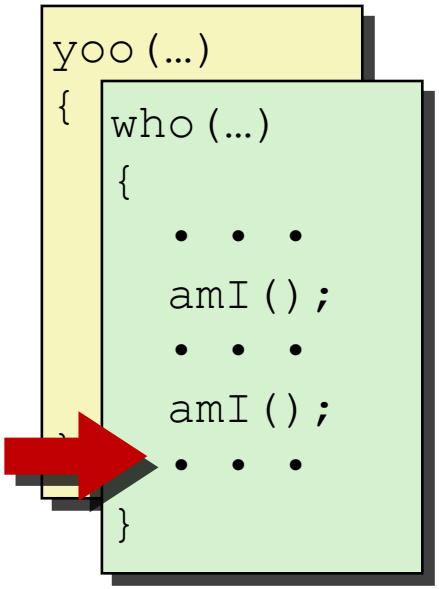
Beispiel: Prozeduraufrufe (cont.)

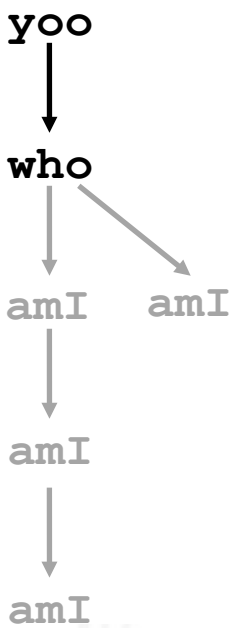


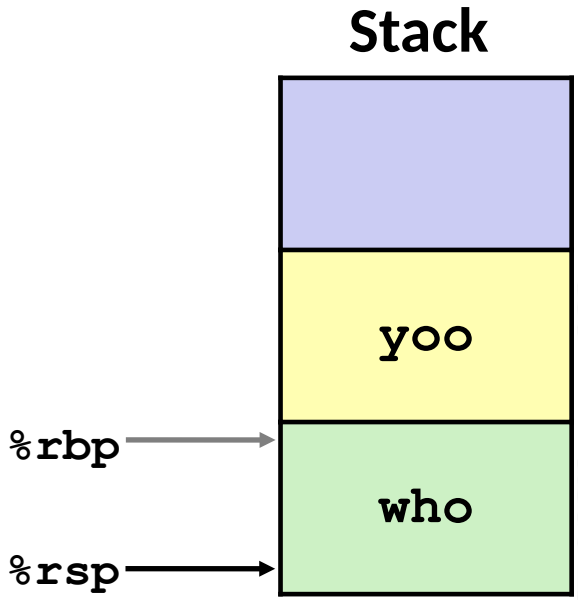




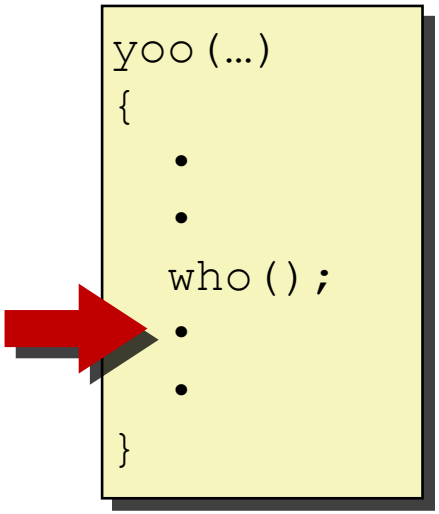
Beispiel: Prozeduraufrufe (cont.)

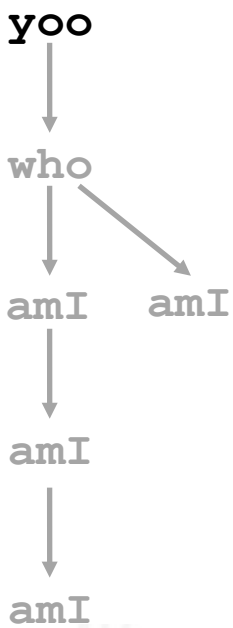


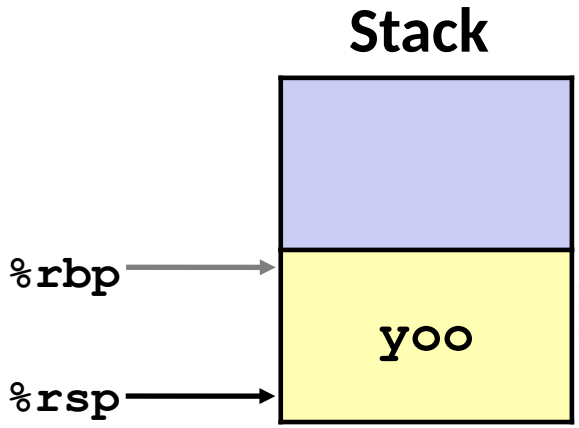




Beispiel: Prozeduraufrufe (cont.)







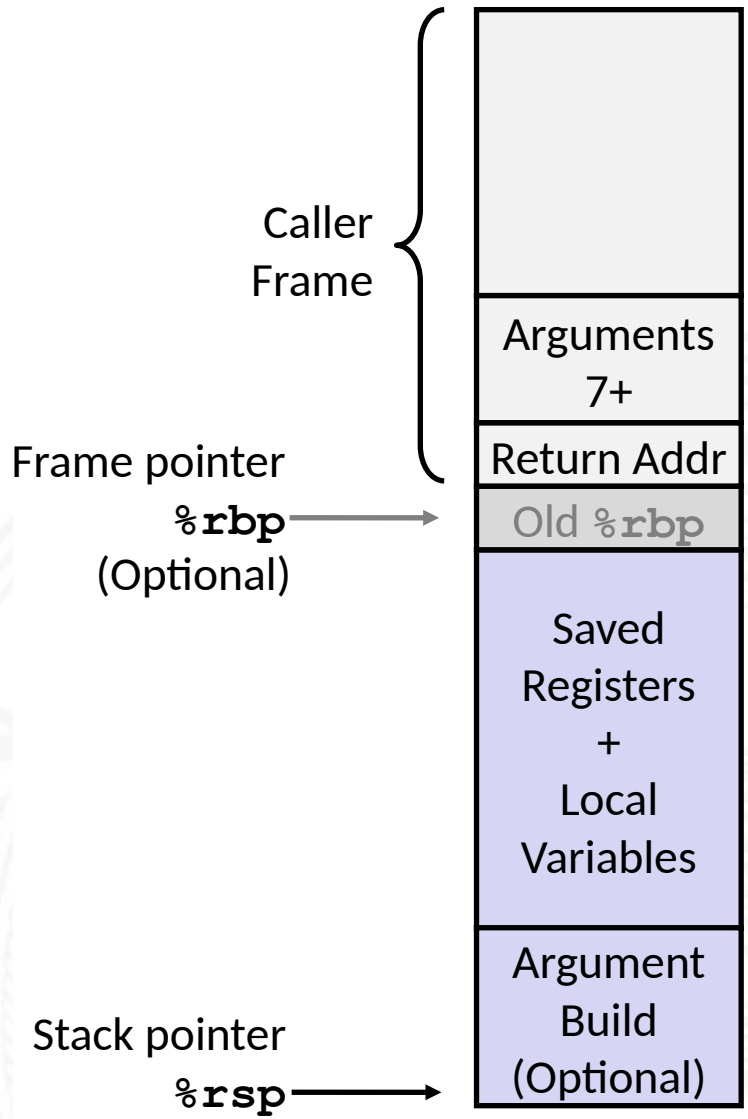
x86-64/Linux Stack-Frame

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

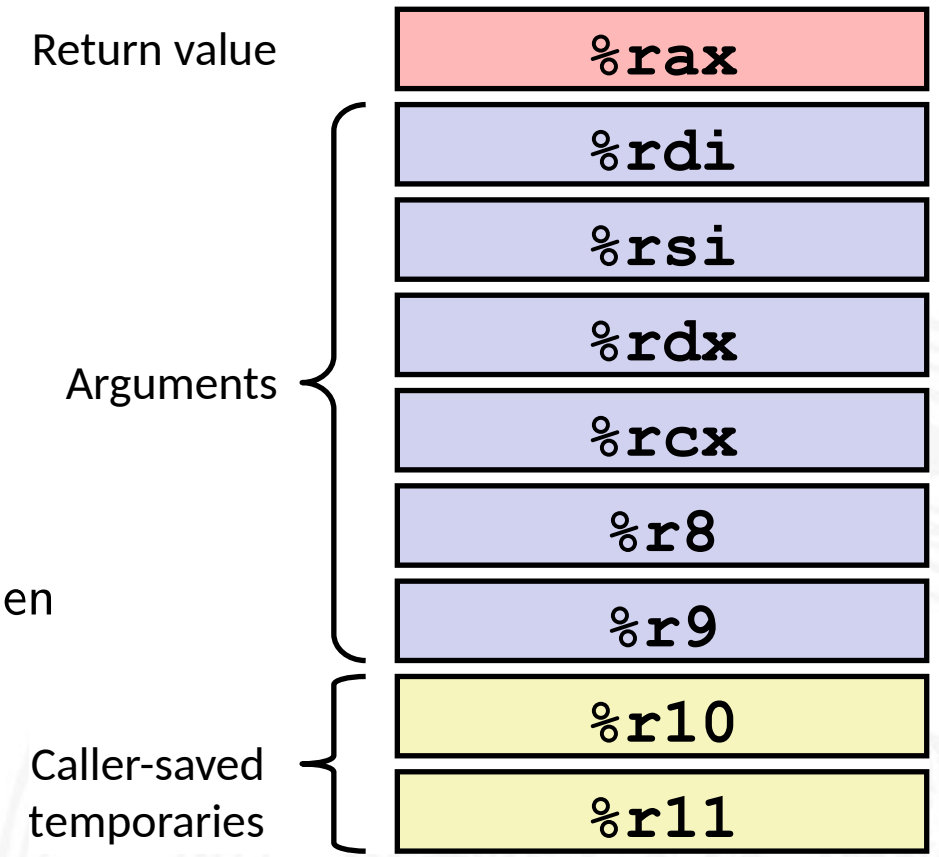
- yoo („Caller“) ruft Prozedur `who` („Callee“) auf

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

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

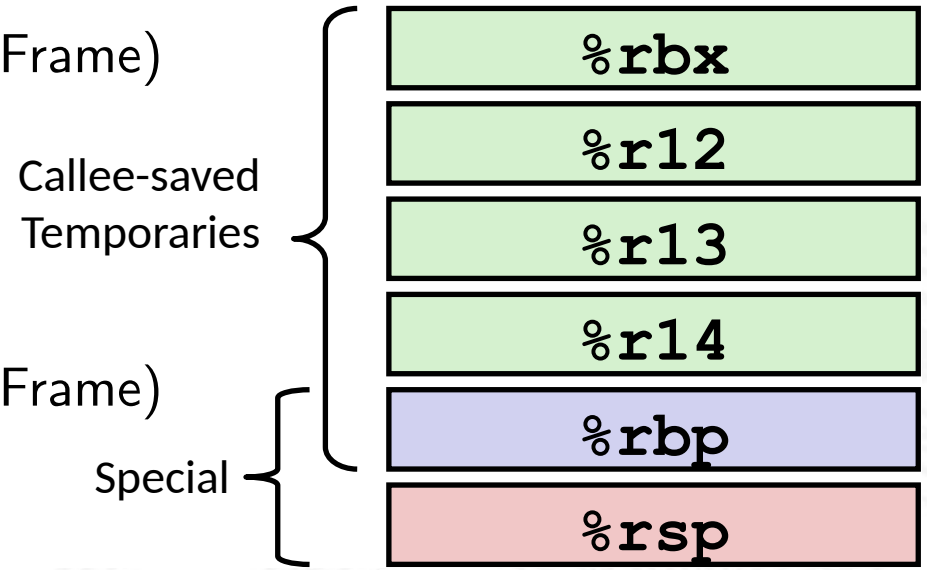
Register Sicherungskonventionen (cont.)

- `%rax`
  - Rückgabewert
  - Caller-Saved
  - kann lokal geschrieben werden
- `%rdi...%r9`
  - Argumente
  - Caller-Saved
  - können lokal geschrieben werden
- `%r10`, `%r11`
  - Caller-Saved
  - können lokal geschrieben werden



Register Sicherungskonventionen (cont.)

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



Elementare Datentypen

- Ganzzahl (Integer)
  - wird in allgemeinen Registern gespeichert
  - abhängig von den Anweisungen: *signed*/*unsigned*
  - Architektur-, Compiler-, OS-abhängig

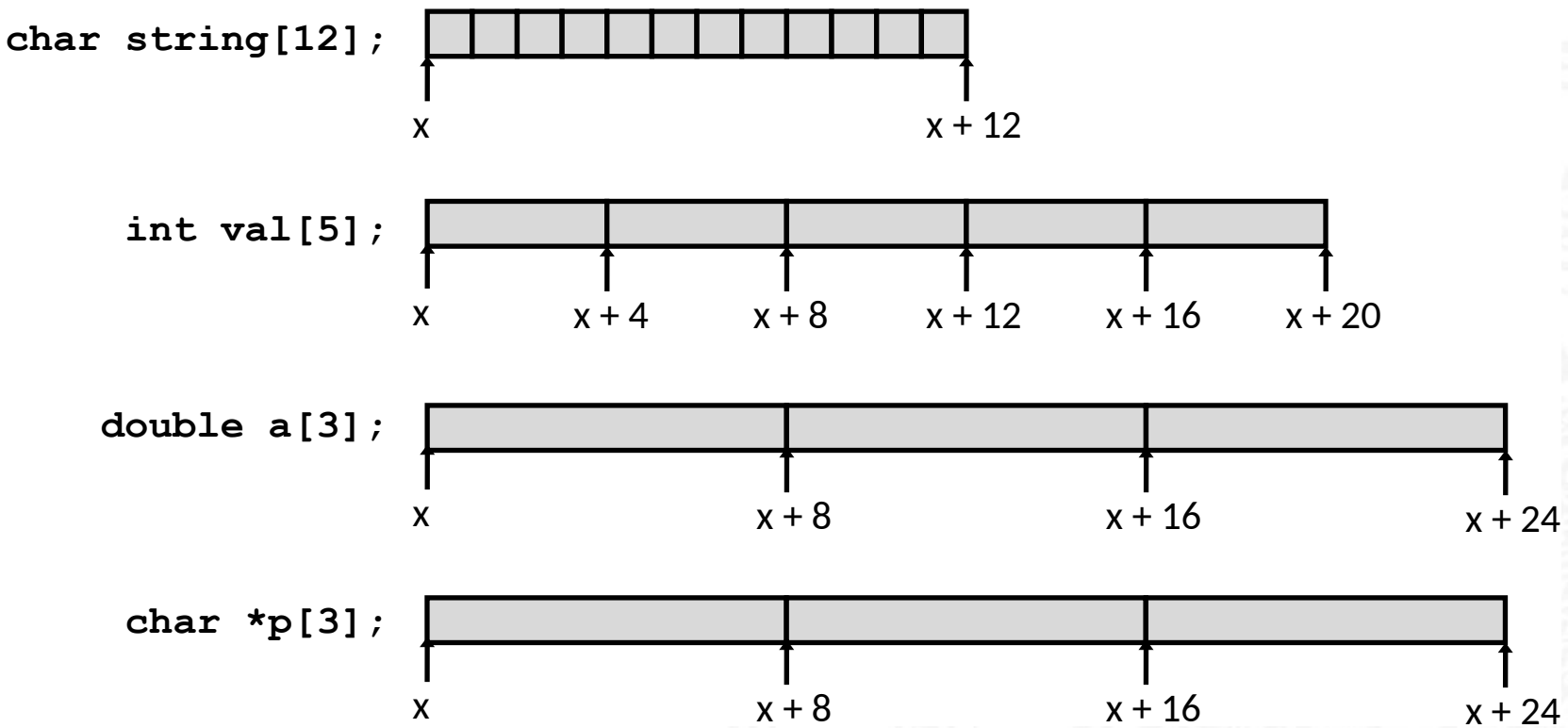
Intel x86	as	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
doubleword	l	4	[unsigned] int / long
quadword	q	8	[unsigned] long / long long

- Gleitkomma (Floating Point)
  - wird in Gleitkomma-Registern gespeichert
  - as: *Gnu ASsembler*

Intel x86	as	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

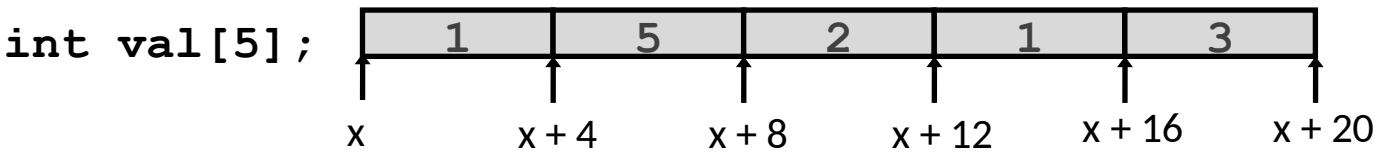
Array: Allokation / Speicherung

- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - fortlaufender Speicherbereich von `N` $\times$ sizeof(`T`) Bytes



Array: Zugriffskonvention

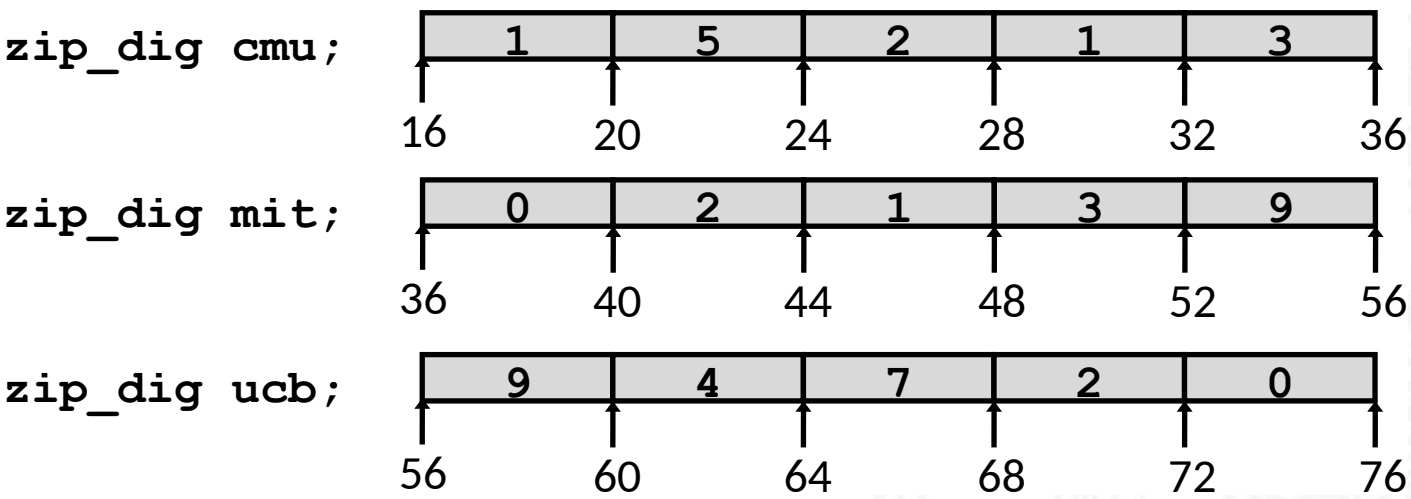
- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - Bezeichner `A` zeigt auf erstes Element des Arrays: Element 0



Reference	Type	Value	
`val[4]`	`int`	`3`	
`val`	`int *`	`x`	
`val+1`	`int *`	`x + 4`	
`&val[2]`	`int *`	`x + 8`	
`val[5]`	`int`	`??`	
`*(val+1)`	`int`	`5`	`//val[1]`
`val + i`	`int *`	`x + 4 * i`	`//&val[i]`

Beispiel: einfacher Arrayzugriff


```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

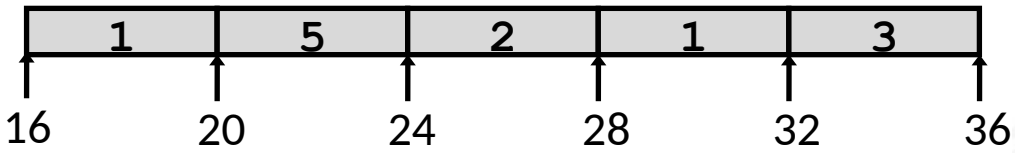


Beispiel: einfacher Arrayzugriff (cont.)

- Adressieren von `%rdi` $+ 4 \times$ `%rsi`

$\Rightarrow$ Speicheradresse `(%rdi,%rsi,4)`

```
zip_dig cmu;
```




```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

Register	Funktion
%rdi	Array Startadresse
%rsi	Array Index

x86-64

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

Beispiel: einfacher Arrayzugriff (cont.)

Achtung bei Arrayzugriffen:

- keine Bereichsüberprüfung („*bounds checking*“)
- Verhalten außerhalb des Indexbereichs ist Implementierungsabhängig

Zwei- und mehrdimensionale Arrays

$(N \times M)$ Matrizen? drei grundsätzliche Möglichkeiten

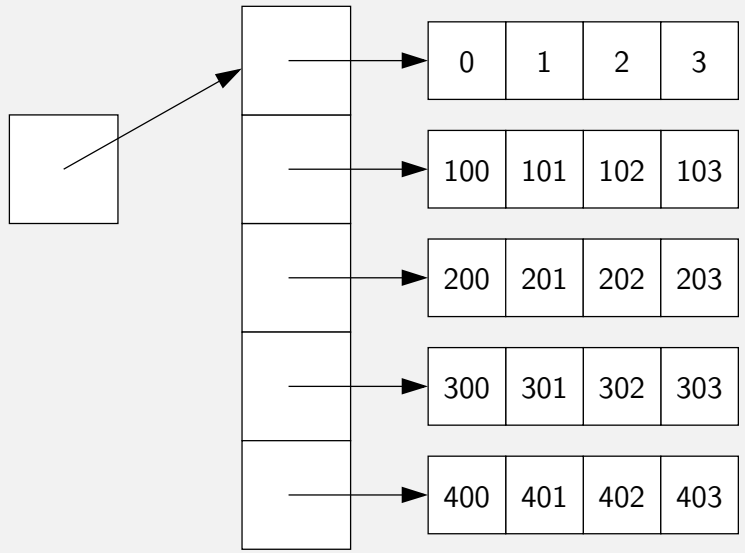
1. Array von Pointern auf Zeilen-Arrays von Elementen — Java
   - sehr flexibel, auch für nicht-rechteckige Layouts
   - Sharing/Aliasing von Zeilen möglich

- Array von $N \times M$ Elementen und passende Adressierung
  2. row-major Anordnung — C, C++
  3. column-major Anordnung — Matlab, FORTRAN

- bei Verwendung/Mischung von Bibliotheksfunktionen aus anderen Sprachen unbedingt berücksichtigen

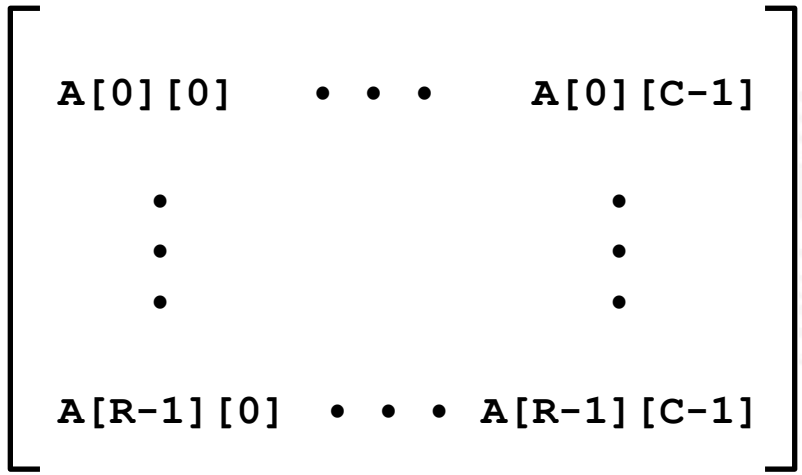
Java: Array von Pointern auf Arrays von Elementen

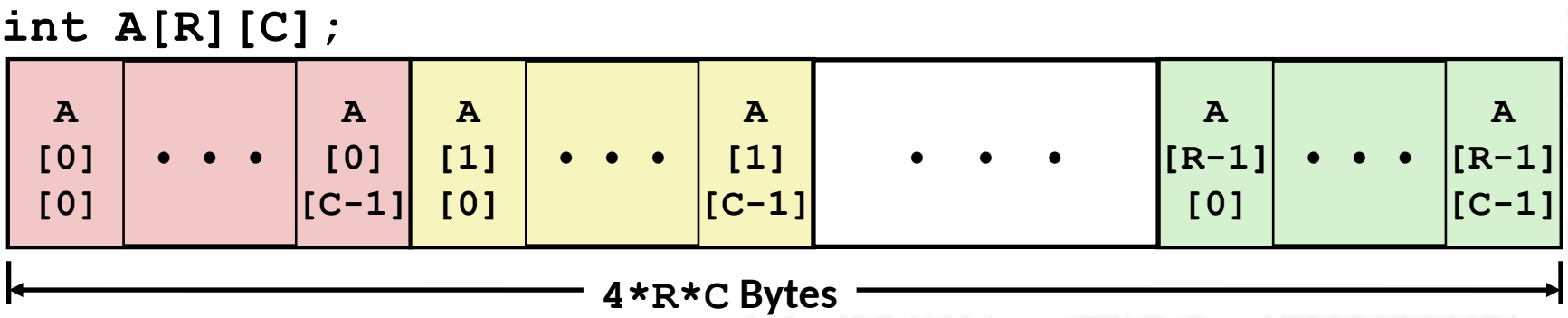

```
class MatrixDemo {
  int matrix[][]; // matrix[i]->

  public MatrixDemo( int NROWS, int NCOLS ) {
    matrix = new int[NROWS][NCOLS];
    for( int r=0; r < matrix.length; r++ ) {
      for( int c =0; c < matrix[r].length; c++ ) {
        matrix[r][c] = 100*r + c;
      }
    }
    // int[] row0 = matrix[0];
    // int    m23 = matrix[2][3];
  }
  public int get( int r, int c ) {
    return matrix[r][c];
  }
}
```



Zweidimensionale Arrays in C

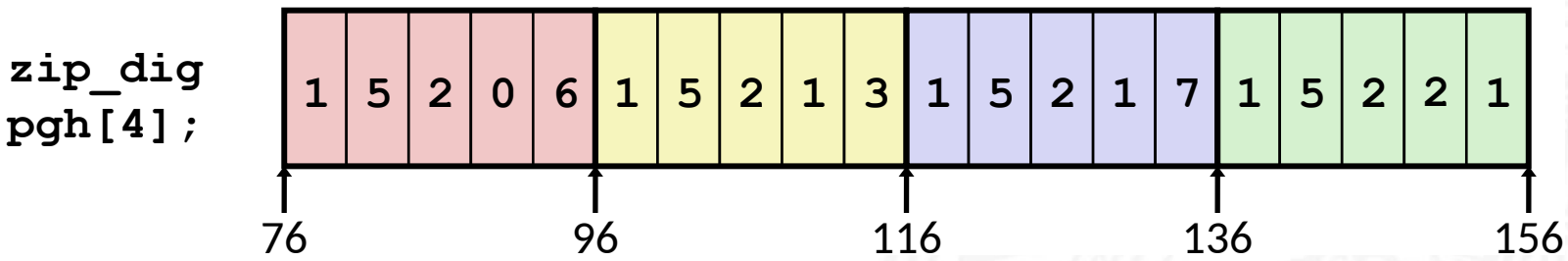
- Deklaration $\langle T \rangle$ $\langle A \rangle$ `[`$\langle R \rangle$`][`$\langle C \rangle$`];`
- Größe: $\langle R \rangle$`*`$\langle C \rangle$`*sizeof(`$\langle T \rangle$`)` Bytes
- „*row-major*“ Anordnung





Zweidimensionale Arrays in C (cont.)


```
#define PCOUNT 4
typedef int zip_dig[5];

zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



Mehrdimensionale Arrays: entsprechend

- d-dimensionales $N_1 \times N_2 \times \cdots \times N_d$ Array
  - Element adressiert mit Tupel $(n_1, n_2, \ldots, n_d)$, mit d (zero-offset) Indizes $n_k \in [0, N - K - 1]$

- row-major Anordnung: letzte Dimension ist fortlaufend

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

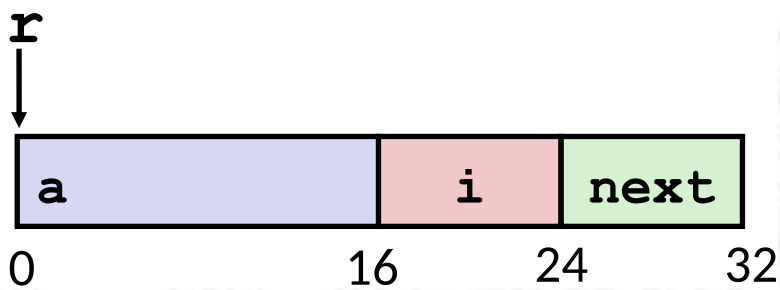
- column-major Anordnung: erste Dimension ist fortlaufend

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots)) = \sum_{k=1}^{d} \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

Strukturen (*Records*)

- Allokation eines zusammenhängenden Speicherbereichs
- Elemente der Struktur über Bezeichner referenziert
- verschiedene Typen der Elemente sind möglich
- Zeiger `r` auf Byte-Array
  - für Zugriff auf Struktur(element)
  - Compiler bestimmt Offset für jedes Element

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

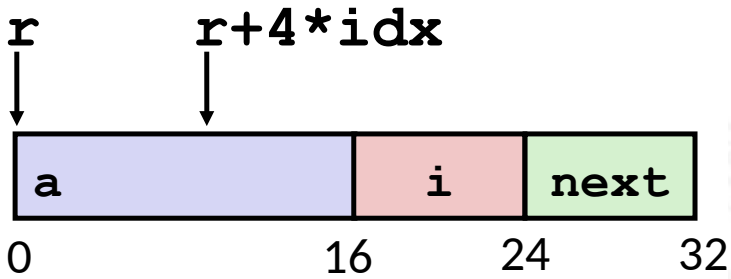


Beispiel: Strukturzugriffe

- Arrayzugriff in Struktur

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

```
int *get_ap
  (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```



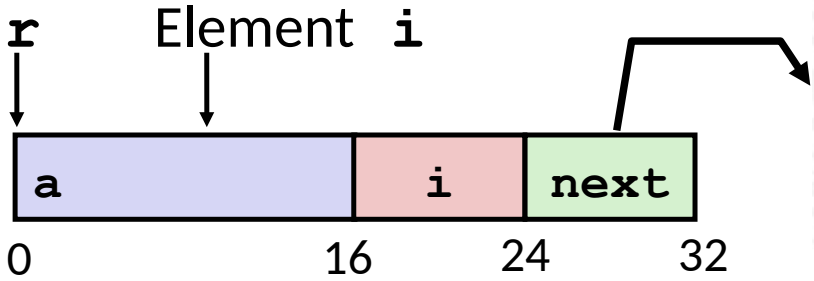

```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Beispiel: Strukturzugriffe (cont.)

- Verlinkte Liste durchlaufen

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




```
.L11:                                   # loop:
  movslq  16(%rdi), %rax                #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)           #   M[r+4*i] = val
  movq    24(%rdi), %rdi                #   r = M[r+24]
  testq   %rdi, %rdi                    #   Test r
  jne     .L11                          #   if !=0 goto loop
```

Ausrichtung der Datenstrukturen (*Alignment*)

- Datenstrukturen an Wortgrenzen ausrichten
  double- / quad-word
- sonst Problem
  - ineffizienter Zugriff über Wortgrenzen hinweg
  - virtueller Speicher und Caching

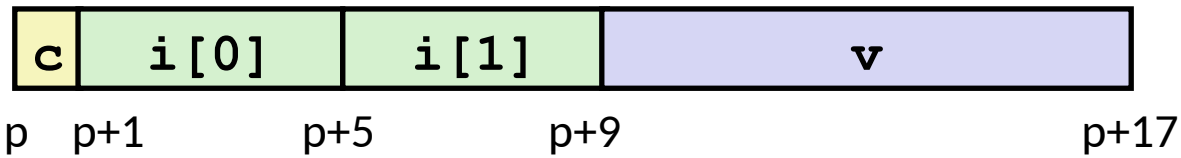
$\Rightarrow$ Compiler erzeugt „Lücken“ zur richtigen Ausrichtung

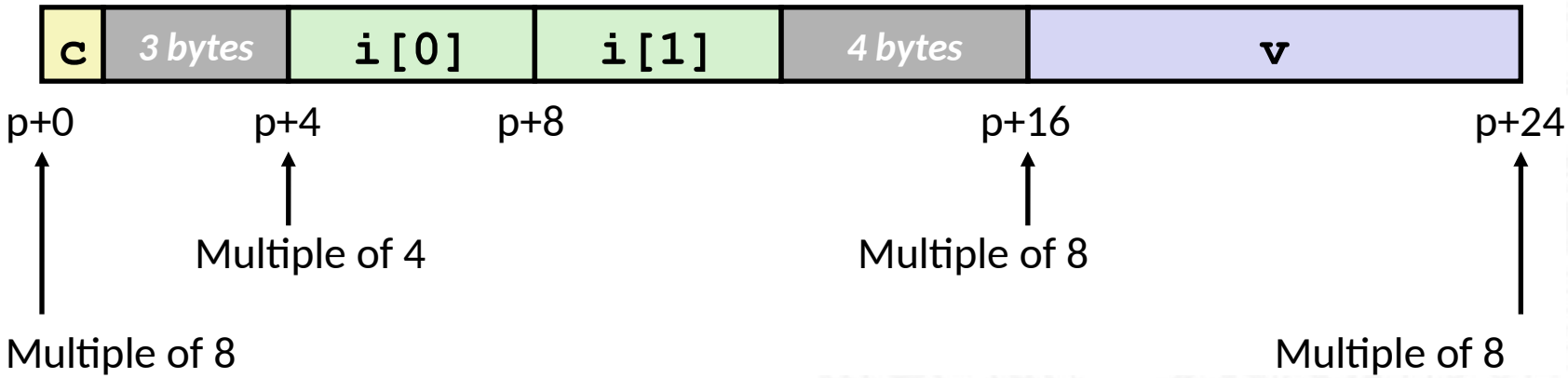
- typisches Alignment (x86-64)

Länge	Typ	Alignment
1 Byte	`char`	keine speziellen Verfahren
2 Byte	`short`	Adressbits: ...0
4 Byte	`int`, `float`	–"– ...00
8 Byte	`double`, `long`, `char *`	–"– ...000

Beispiel: Structure Alignment


```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```





Linker und Loader

- Programm in mehrere Quelldateien aufgeteilt

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

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

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

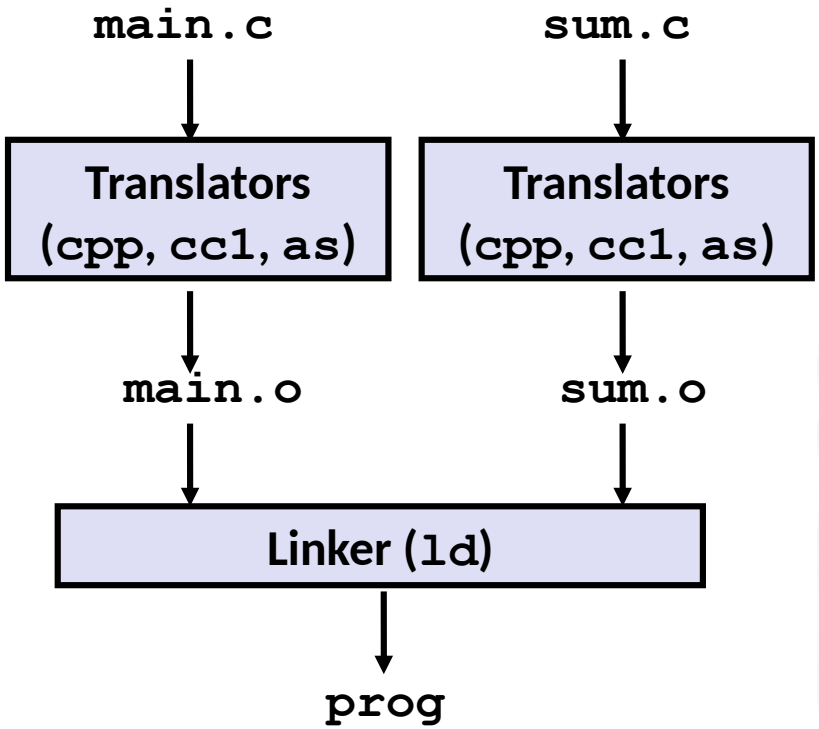
main.c

```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 Parameter für jedes Teilprogramm `man gcc`



Linker und Loader (cont.)

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

$\Rightarrow$ vorgefertigte Programmbibliotheken

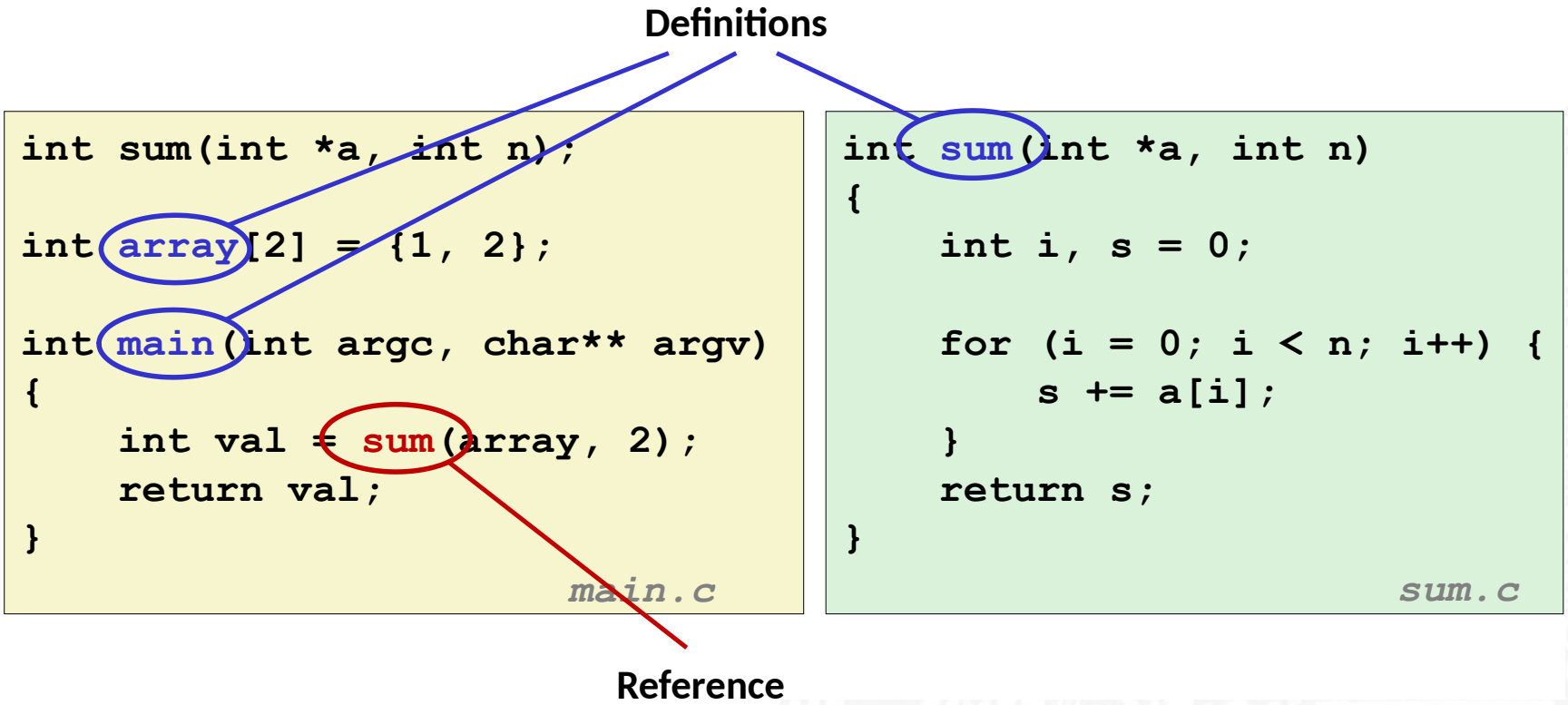
\+ Effizienz

$\Rightarrow$ Zeitvorteil
- nach Änderung müssen nur kleine Teile neu übersetzt werden
- ermöglicht paralleles Compilieren

$\Rightarrow$ (Speicher-) Platzvorteil
- wichtige Funktionen in Datei aggregiert (z.B. `malloc`, `printf`)
- ermöglicht gemeinsame Nutzung

Aufgaben des Linkers

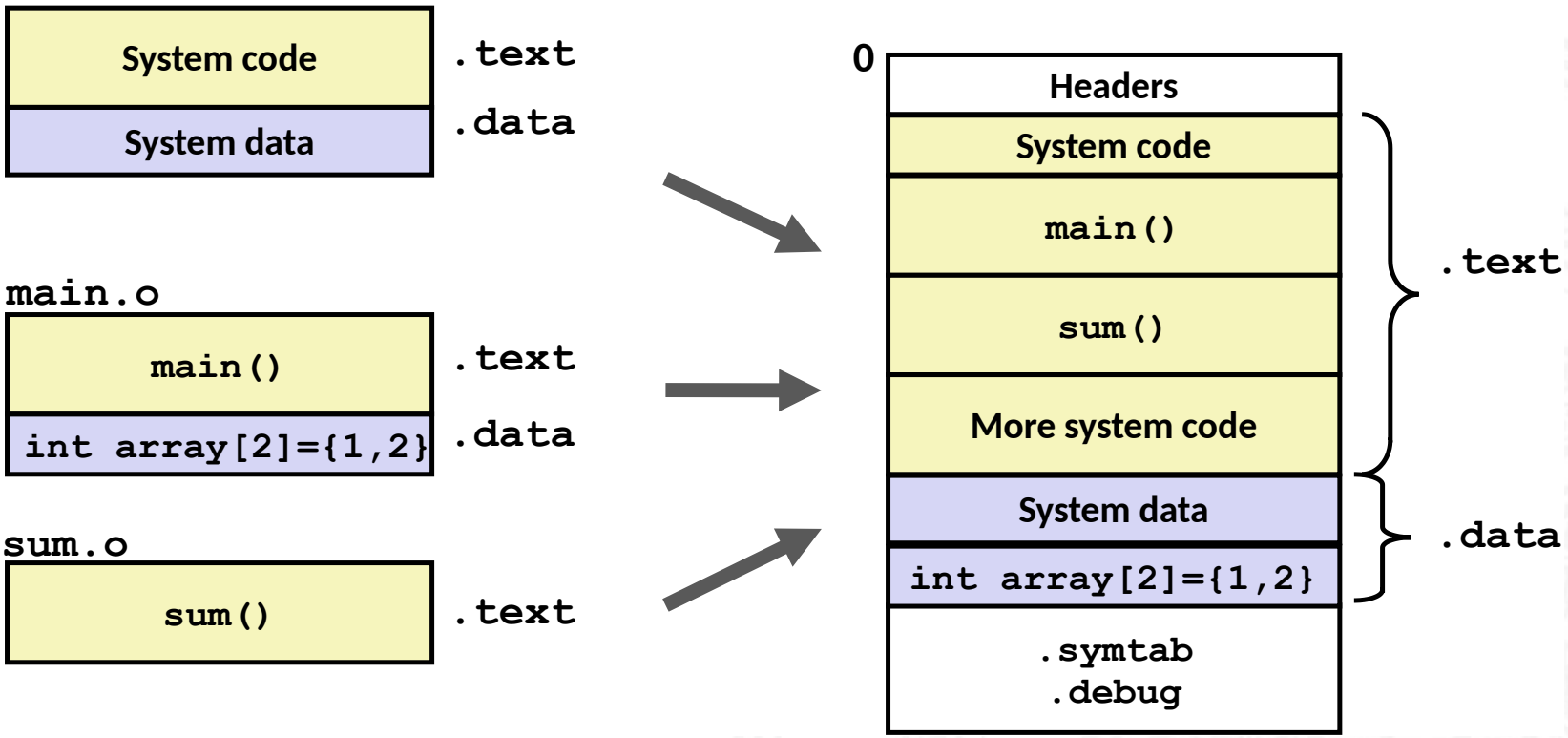
1. Symbole identifizieren (globale Variablen, Funktionen)
   Symbole auflösen (= eindeutig machen) ⇒ Symboltabelle



Aufgaben des Linkers (cont.)

2. „Relocation“
   - Programmcode und -daten der Quelldateien zusammenfassen
   - Symboltabellen zusammenfassen

   ⇒ Speicheradressen eindeutig machen: Sprünge+Symbole



⇒ erzeugt ein ausführbares Programm für *Loader*

Statisches / dynamisches Linken

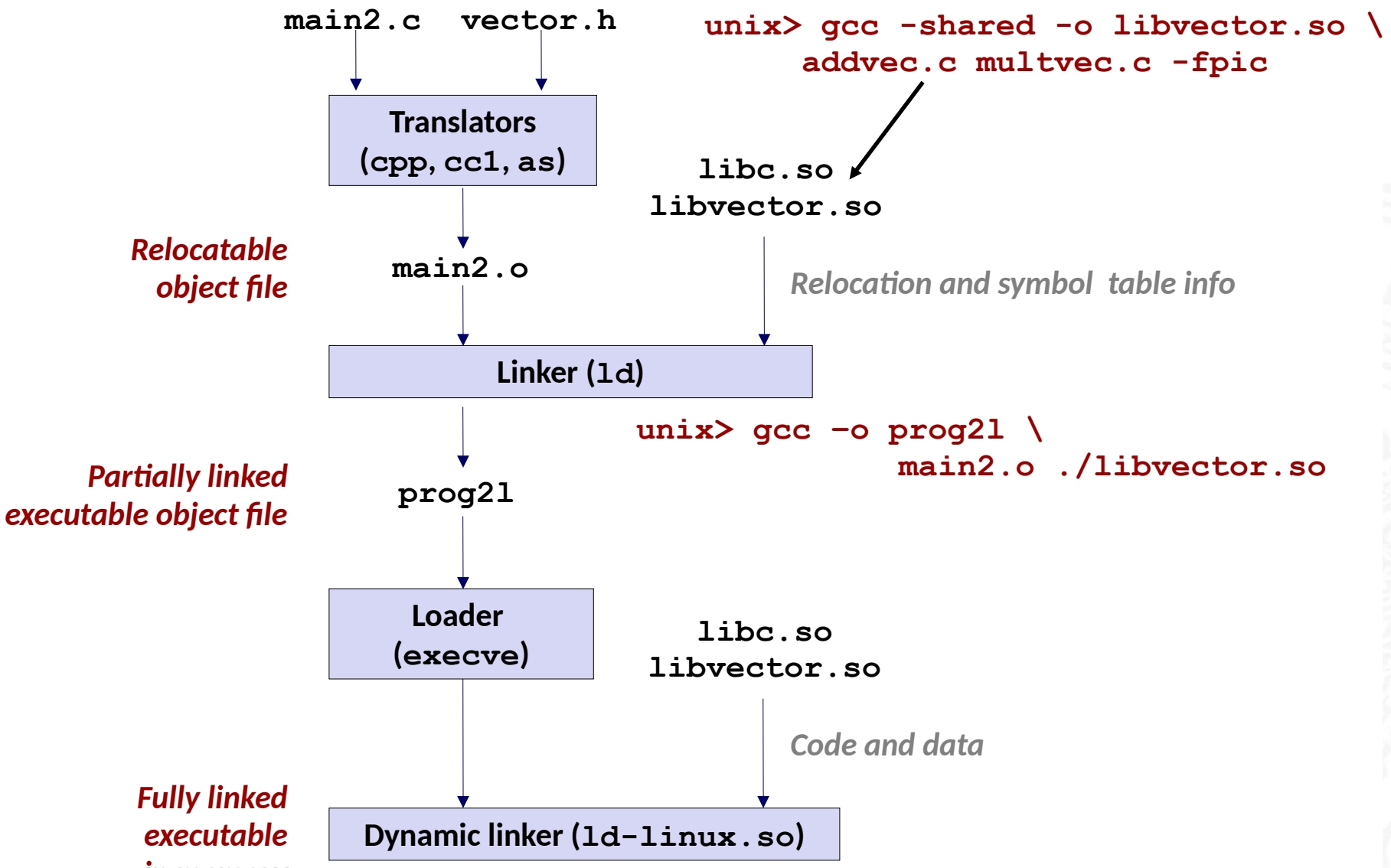
- Statisches Binden („static linking“)
  - Funktionen aus Bibliotheksarchiven (`.a`-Dateien) werden in ausführbares Programm eingebaut
  - nicht genutzte Funktionen werden entfernt
  - Linken während Compilierung

- Dynamisches Binden („dynamic linking“)
  - Bibliotheken werden erst beim Laden in Speicher oder sogar 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



Assembler-Programmierung

viele Themen aus Zeitgründen nicht behandelt

- Linker und Loader
  - genauere Funktionsweise von Linker und Loader
  - programmiertechnische Realisierung
  - Probleme bei der Symbolauflösung
- Speicherverwaltung
  - 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 978–3–8689–4238–5

[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.)

[PH20] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – MIPS Edition*. 6th edition, Morgan Kaufmann Publishers Inc., 2020. ISBN 978–0–12–820109–1

[PH16] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*. 5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0

[Hyd10] R. Hyde: *The Art of Assembly Language Programming*. 2nd edition, No Starch Press, 2010. ISBN 978–1–59327–207–4. `www.plantation-productions.com/Webster/www.artofasm.com`