64-040 Modul IP7: Rechnerstrukturen

11 Assemblerprogrammierung

Norman Hendrich

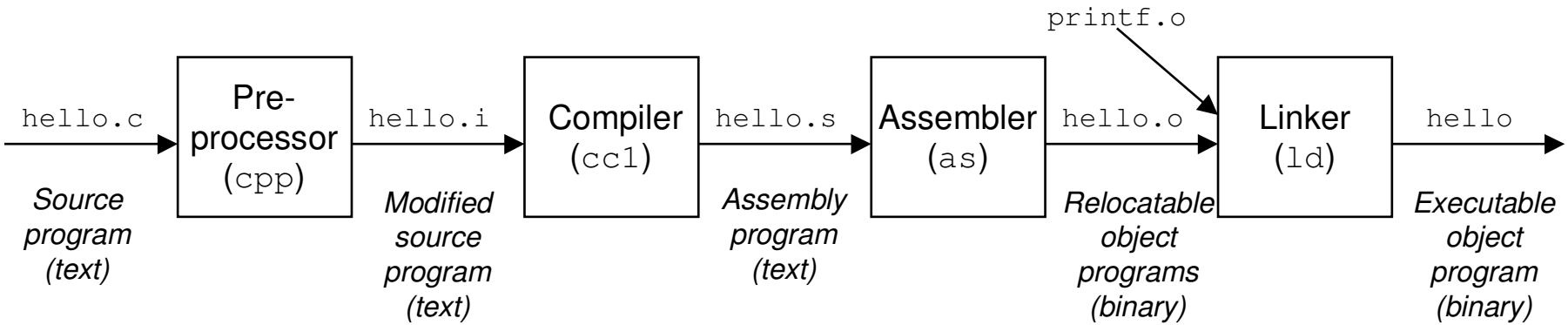
Universität Hamburg
MIN Fakultät, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

WS 2013/2014

Inhalt

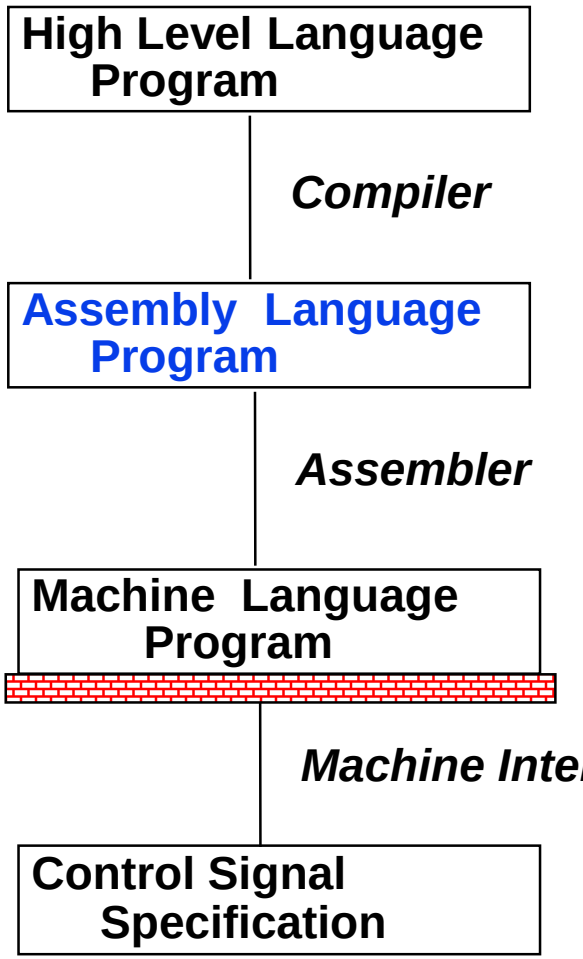
1. Assembler-Programmierung
   - Grundlagen der Assemblerebene
   - x86 Assemblerprogrammierung
   - Elementare Befehle und Adressierungsarten
   - Kontrollfluss
   - Sprungbefehle und Schleifen
   - Mehrfachverzweigung (Switch)
2. Funktionsaufrufe und Stack
   - push- und pop-Befehle
   - call- und ret-Befehle
   - Stack-basierende Programmierung
   - Aufruf-Konventionen: caller-save und callee-save

Wiederholung: Abstraktionsebenen



- verschiedene Repräsentationen eines Programms
  - Hochsprache
  - Assembler
  - Maschinensprache
- Maschinensprache wird dann ausgeführt
  - von-Neumann Zyklus: Befehl holen, dekodieren, ausführen
  - reale oder virtuelle Maschine

Wiederholung: Abstraktionsebenen (cont.)




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,   0($2)
lw  $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK

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
- Programmleistung verbessern
  - Ursachen für ineffiziente Programme verstehen
  - "maschinengerechte" Datenstrukturen / Algorithmen
- Systemsoftware implementieren
  - Compilerbau: Maschinencode als Ziel
  - Gerätetreiber
  - Betriebssysteme

Assembler: Lernziele

- Grundverständnis der Programmausführung
  - Umsetzung arithmetisch/logischer Operationen
  - Umsetzung der gängigen Kontrollstrukturen: Bedingte Sprünge, Schleifen, Switch
  - Datenstrukturen
  - ein- und mehrdimensionale Arrays
- Funktionsaufrufe — Stack
  - Funktionsparameter — by-value, by-reference
  - lokale Variablen
  - rekursive Funktionen
- Grundlagen der Speicherverwaltung — Heap
- Umsetzung objektorientierter Konzepte im Rechner — vtable

Assembler: Speicherverwaltung

- Speicher aufgeteilt in mehrere Regionen
  - Programmcode
  - Funktionsbibliotheken, Linker und Loader
  - Stack mit Funktionsaufrufen und lokalen Variablen
  - statisch allozierte Daten und globale Variablen
  - dynamisch allozierte Daten
  - Umsetzung objektorientierter Konzepte

- Programmierfehler und Sicherheitslücken
  - aktuelle Rechner bieten keinen/kaum Speicherschutz
  - geschützte Systeme ("capabilities") bisher am Markt gescheitert

  - fehlerhafte dynamische Speicherverwaltung
  - Pufferüberläufe, Stack-allocated Daten
  - Ausnutzen durch bösartigen Code

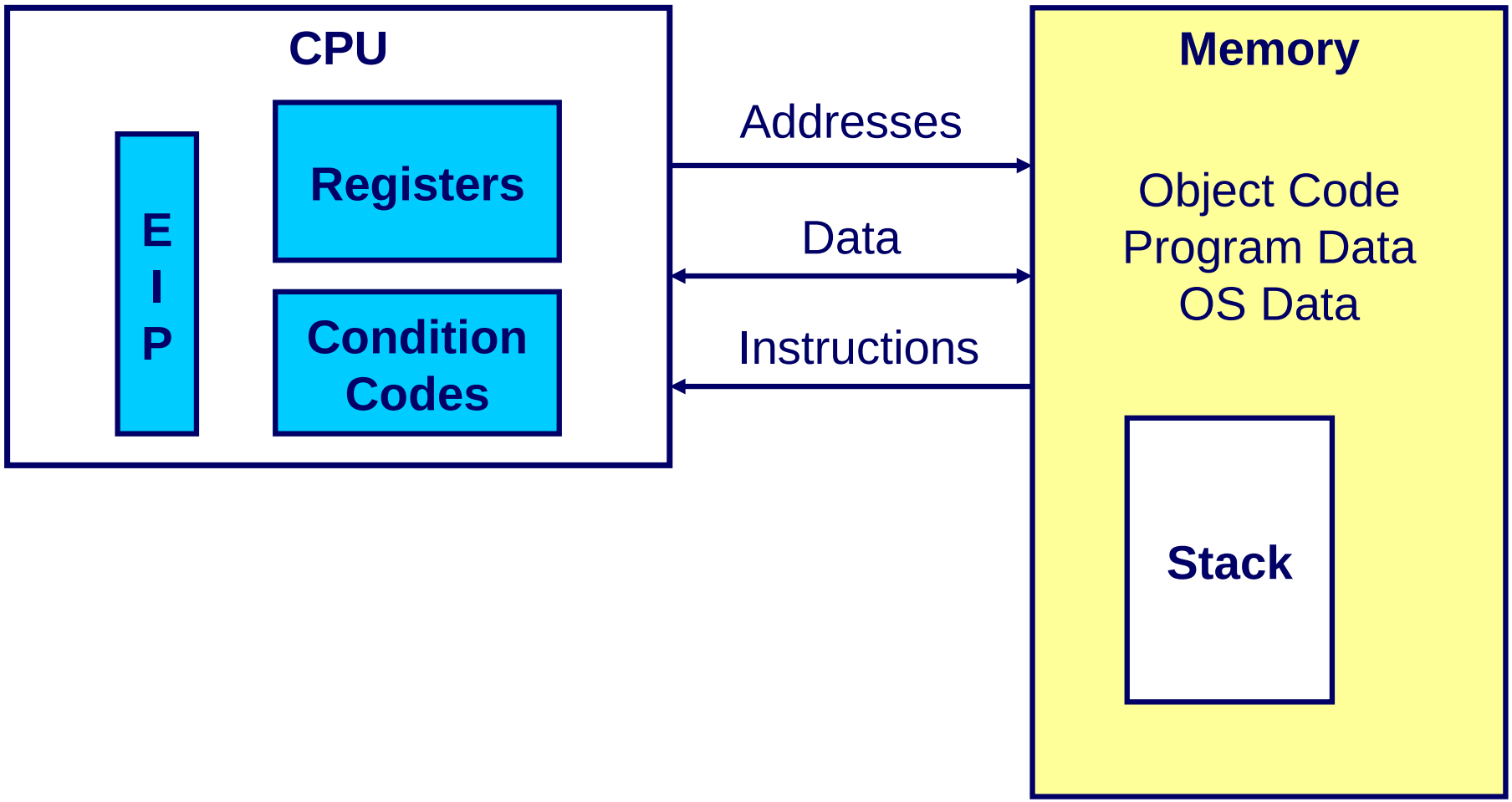
Assembler

- Beschränkung auf wesentliche Konzepte
  - GNU Assembler für x86 (32-bit)
  - nur ein Datentyp: 32-bit Integer (long)
  - nur kleiner Subset des gesamten Befehlssatzes

- diverse nicht behandelte Themen:
  - Makros
  - Implementierung eines Assemblers (2-pass)
  - Tips für effizientes Programmieren
  - Befehle für die Systemprogrammierung (supervisor mode)
  - x86 Gleitkommabefehle
  - ...

Assembler-Programmierung

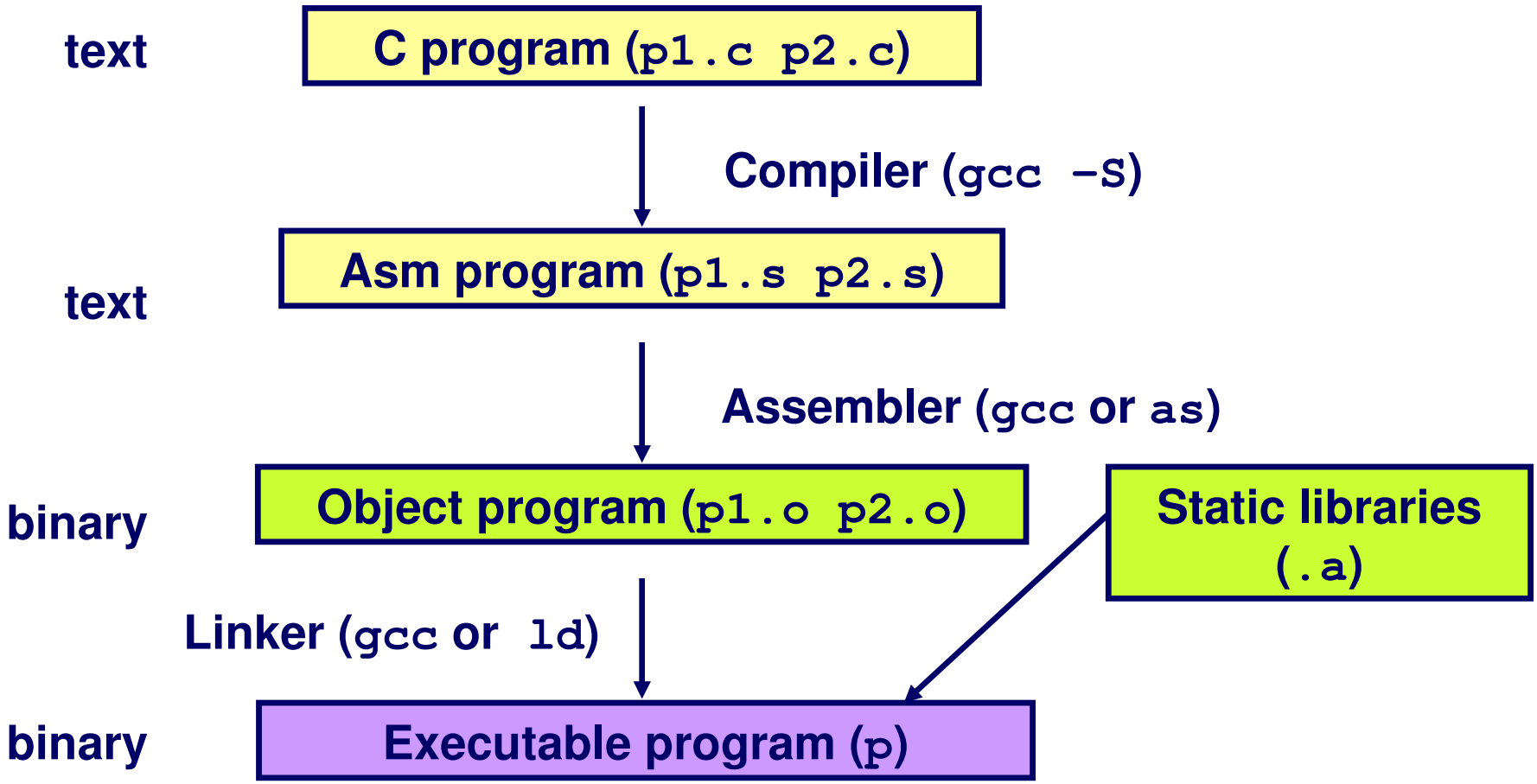
Rechner aus Sicht des Programmierers



Beobachtbare Zustände (Assemblerebene)

- Programmzähler (*Instruction Pointer*) — x86 EIP Register
  - Adresse der nächsten Anweisung
- Registerbank — EAX..EBP Register
  - häufig benutzte Programmdaten
- Zustandscodes — EFLAGS Register
  - Statusinformationen des Prozessors,
  - u.a. über die letzte arithmetische Operation
  - für bedingte Sprünge benötigt (*Conditional Branch*)
- Speicher
  - byteweise adressierbares Array
  - Programmcode, Nutzerdaten, Betriebssystem-Daten
  - beinhaltet den Stack (Kellerspeicher) mit Funktionsaufrufen

Umwandlung von C in Objektcode



Beispiel: Funktion sum()

code.c

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

code.s

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

- Befehl `gcc -O -S code.c`
- Erzeugt `code.s`

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
  - Addressen für alle verfügbaren Adressierungsarten
  - Konvention bei gcc/gas 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 Adressenverweise

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 sum()

- 13 bytes Programmcode
- x86-Befehle mit 1-, 2- oder 3 bytes
- Erklärung s.u.

- Startadresse: `0x401040`
- vom Compiler/Assembler gewählt

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Assembler und Linker

Assembler

- übersetzt `.s` zu `.o`
- binäre Codierung jeder Anweisung
- (fast) vollständiges Bild des ausführbaren Codes
- Verknüpfungen zwischen Code in verschiedenen Dateien fehlen

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 zur Laufzeit erstellt

Beispiel: Maschinenbefehl für Addition

- C-Code
  - addiert zwei Ganzzahlen mit Vorzeichen

```
int t = x+y;
```

- Assembler
  - Addiere zwei 4-byte Integer
    - `long` Wörter (für gcc)
    - keine signed/unsigned Unterscheidung
  - Operanden

    `x:` Register `%eax`
    `y:` Speicher `M[%ebp+8]`
    `t:` Register `%eax`
    Ergebnis in `%eax`

```
addl 8(%ebp),%eax
```

Similar to expression

```
x += y
```

- Objektcode (laut x86-Befehlssatz)
  - 3-Byte Befehl
  - Speicheradresse `0x401046`

```
0x401046:    03 45 08
```

Objektcode Disassembler: `objdump`

```
00401040 <_sum>:
   0:       55                      push   %ebp
   1:       89 e5                   mov    %esp,%ebp
   3:       8b 45 0c                mov    0xc(%ebp),%eax
   6:       03 45 08                add    0x8(%ebp),%eax
   9:       89 ec                   mov    %ebp,%esp
   b:       5d                      pop    %ebp
   c:       c3                      ret
   d:       8d 76 00                lea    0x0(%esi),%esi
```

- `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

Alternativer Disassembler: gdb

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

gdb Debugger

`gdb p`

`disassemble sum`

- **Disassemble procedure**

`x/13b sum`

- **Examine the 13 bytes starting at `sum`**

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:  55                 push   %ebp
30001001:  8b ec              mov    %esp,%ebp
30001003:  6a ff              push   $0xffffffff
30001005:  68 90 10 00 30     push   $0x30001090
3000100a:  68 91 dc 4c 30     push   $0x304cdc91
```

- 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

- alle Beispiele und Grafiken dieses Abschnitts sind aus: *Bryant/O'Hallaron: Computer systems – A programmers perspective*
- Beispiele nutzen nur die 32-bit (long) Datentypen
  - x86 wird wie 8-Register 32-bit Maschine benutzt (=RISC)
  - CISC Komplexität und Tricks bewusst vermieden
- Beispiele nutzen gcc/gas Syntax (vs. Microsoft/Intel)

`movl`: Befehl für Datentransfer

- Format: `movl` ⟨*src*⟩`,` ⟨*dst*⟩
- transferiert ein 4-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: 8 Ganzzahl-Register
    - `%esp` und `%ebp` für spezielle Aufgaben reserviert (s.u.)
    - z.T. Spezialregister für andere Anweisungen
  - Speicher: 4 konsekutive Speicherbytes
    - zahlreiche Adressmodi

movl: Operanden-Kombinationen

	Source	Destination		C Analogon
movl	Imm	Reg	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	Reg	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	Reg	`movl (%eax),%edx`	`temp = *p;`

`movl`: Operanden/Adressierungsarten

- Immediate: `$x` → `x`
  - positiver (oder negativer) Integerwert
- Register:
  - Inhalt eines der 8 Universalregister, EAX..EBP
- Normal: `(R)` → `Mem[Reg[R]]`
  - Register `R` spezifiziert die Speicheradresse
  - Beispiel: `movl (%ecx), %eax`
- Displacement: `D(R)` → `Mem[Reg[R]+D]`
  - Register `R`
  - Konstantes „Displacement“ D spezifiziert den „offset“
  - Beispiel: `movl 8(%ebp), %edx`

Beispiel: Funktion swap()

Nutzung der Register: ecx: yp, edx: xp, eax: t1, ebx: t0

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

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

x86: Indizierte Adressierung

- allgemeine Form
  - `Imm(Rb,Ri,S)` → `Mem[Reg[Rb]+S*Reg[Ri]+Imm]`
    - ⟨*Imm*⟩ Offset
    - ⟨*Rb*⟩ Basisregister: eins der 8 Integer-Registern
    - ⟨*Ri*⟩ Indexregister: jedes außer %esp
      %ebp 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]]`

x86: Beispiele für Adressberechnung

%edx	0xf000
%ecx	0x100

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

x86: Arithmetische Operationen

Unäre Operatoren

Format	Computation
`incl` ***Dest***	***Dest*** `=` ***Dest*** `+ 1`
`decl` ***Dest***	***Dest*** `=` ***Dest*** `– 1`
`negl` ***Dest***	***Dest*** `= –` ***Dest***
`notl` ***Dest***	***Dest*** `= ~` ***Dest***

x86: Arithmetische Operationen

Binäre Operatoren

Format	Computation	
`addl` *Src,Dest*	*Dest* = *Dest* + *Src*	
`subl` *Src,Dest*	*Dest* = *Dest* – *Src*	
`imull` *Src,Dest*	*Dest* = *Dest* * *Src*	
`sall` *Src,Dest*	*Dest* = *Dest* << *Src*	also called `shll`
`sarl` *Src,Dest*	*Dest* = *Dest* >> *Src*	Arithmetic
`shrl` *Src,Dest*	*Dest* = *Dest* >> *Src*	Logical
`xorl` *Src,Dest*	*Dest* = *Dest* ^ *Src*	
`andl` *Src,Dest*	*Dest* = *Dest* & *Src*	
`orl` *Src,Dest*	*Dest* = *Dest* | *Src*	

Beispiel: arithmetische Operationen

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

```
arith:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

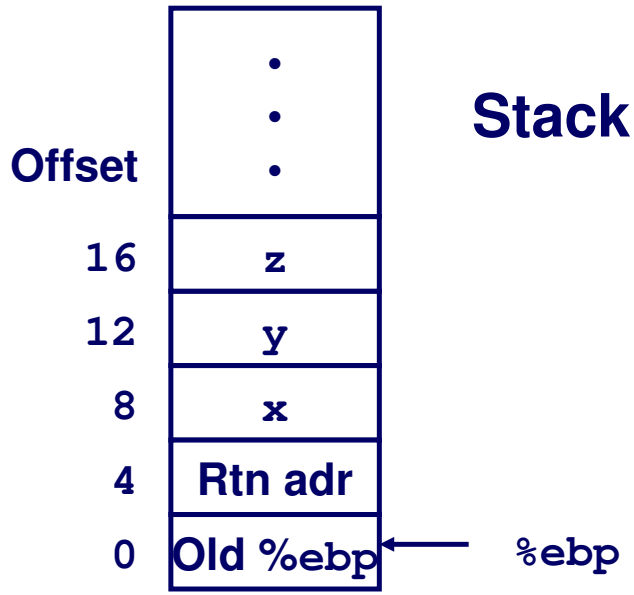
Set Up

Body

Finish

Beispiel: arithmetische Operationen (cont.)

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




```
movl 8(%ebp),%eax        # eax = x
movl 12(%ebp),%edx       # edx = y
leal (%edx,%eax),%ecx    # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx             # edx = 48*y (t4)
addl 16(%ebp),%ecx       # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax   # eax = 4+t4+x (t5)
imull %ecx,%eax          # eax = t5*t2 (rval)
```

x86: `leal`-Befehl: load effective address

- performs the address calculation for a memory access
  - `Imm(Rb,Ri,S)` $\rightarrow$ `Mem[Reg[Rb]+S*Reg[Ri]+Imm]`
- but result is saved in a register (without memory access)

- very useful for general arithmetic
- often used by compilers, see above example

Beispiel: logische Operationen

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

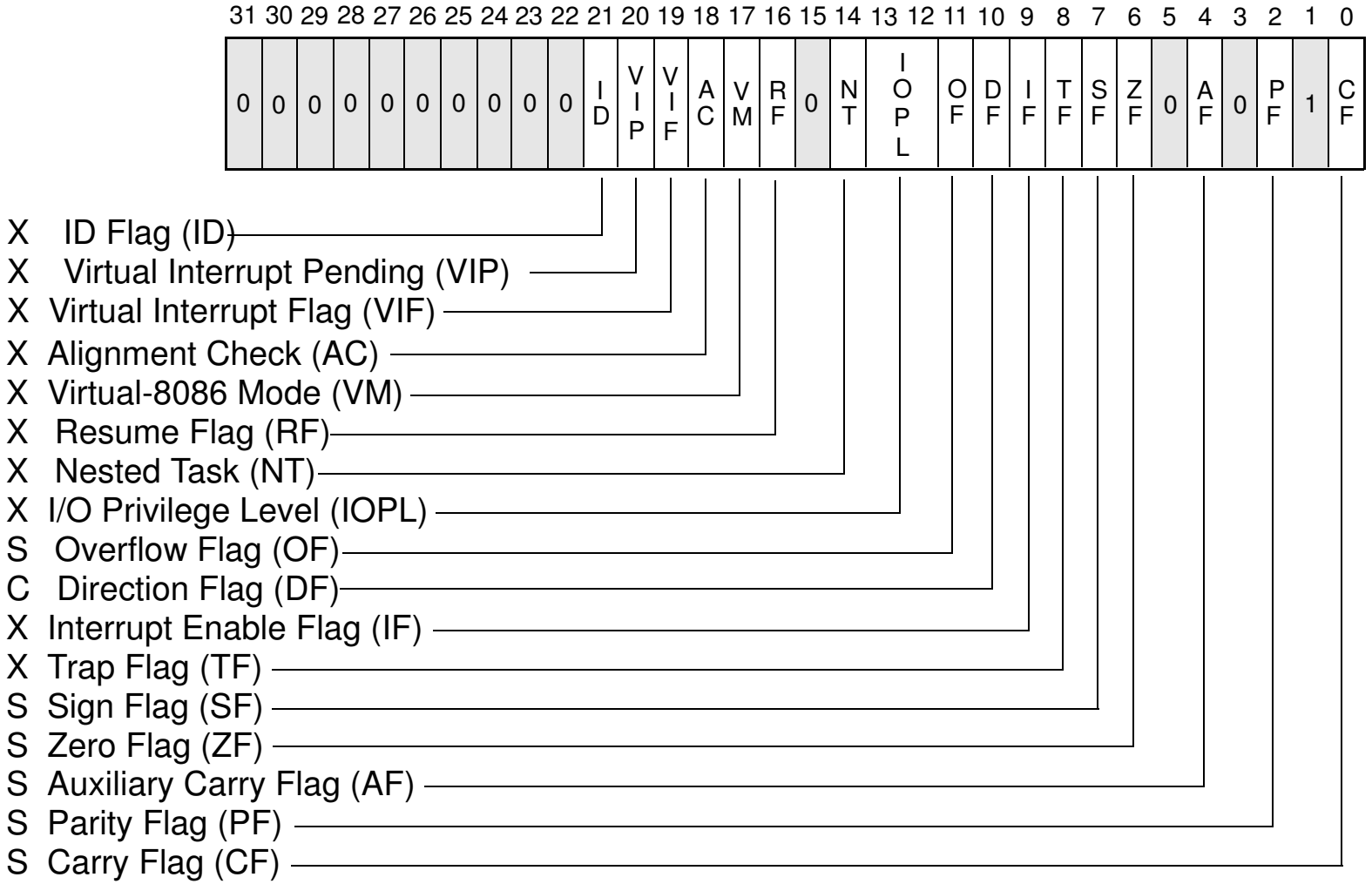
Finish

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y      (t1)
sarl $17,%eax          eax = t1>>17   (t2)
andl $8185,%eax        eax = t2 & 8185
```

Kontrollfluss / Programmstrukturen

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

x86: EFLAGS Register



Zustandscodes

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

1. implizite Aktualisierung durch arithmetische Operationen
   - Beispiel: `addl` $\langle src \rangle, \langle dst \rangle$ 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: `cmpl` $\langle src2 \rangle, \langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle - \langle src2 \rangle$ (`subl` $\langle src2 \rangle, \langle src1 \rangle$)
     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
     $(a > 0 \ \wedge b < 0 \ \wedge (a - b) < 0) \ ||$
     $(a < 0 \ \wedge b > 0 \ \wedge (a - b) \geq 0)$

Zustandscodes (cont.)

3. explizites Setzen durch `testl`-Befehl
   - Beispiel: `testl` $\langle src2 \rangle, \langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle \&\& \langle src2 \rangle$ (`andl` $\langle src2 \rangle, \langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats
   - hilfreich, wenn einer der Operanden eine Bitmaske ist

   - `ZF` setzen wenn $(src1 \wedge src2) = 0$
   - `SF` setzen wenn $(src1 \wedge src2) < 0$

set..-Befehle: Zustandscodes in Register übertragen

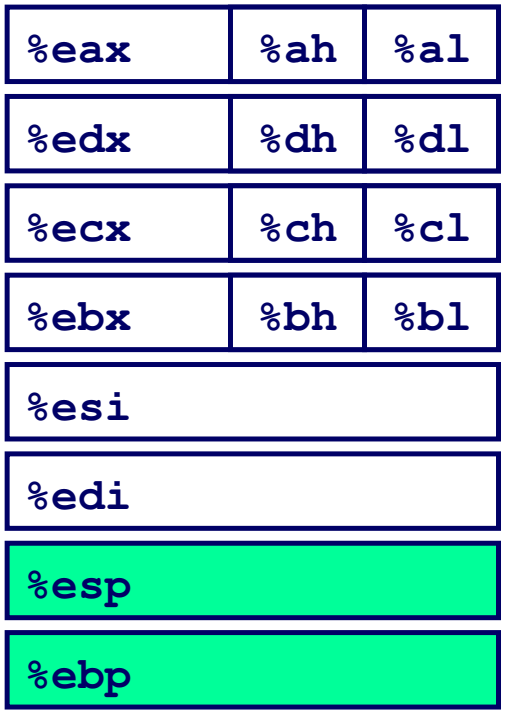
- Befehle setzen einzelnes Byte (LSB) in den Universalregistern, basierend auf den ausgewählten Zustandscodes

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)
- oft kombiniert mit `movzbl`

  „move with zero-extend byte to long“,
  also dem Löschen der Bits 31..8




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

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```

Sprungbefehle („Jump“)

j.. Anweisungen

- 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: Labels

- Assemblercode enthält je einen Maschinenbefehl pro Zeile
- normale Programmausführung ist sequentiell
- Befehle beginnen an eindeutig bestimmten Speicheradressen

- Labels: symbolische Namen für bestimmte Adressen
  - am Beginn einer Zeile, oder vor einem Befehl
  - vom Programmierer / Compiler vergeben
  - Verwendung als symbolische Adressen für Sprünge

  - `_max`: global, Beginn der Funktion `max()`
  - `L9:` lokal, nur vom Assembler verwendete interne Adresse

  - Labels müssen in einem Programm eindeutig sein

x86: bedingter Sprung („Conditional Branch“)

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

x86: bedingter Sprung („Conditional Branch“)

- C-Code erlaubt goto
- entspricht mehr dem Assemblerprogramm
- schlechter Programmierstil

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

```
    movl 8(%ebp),%edx    # edx = x
    movl 12(%ebp),%eax   # eax = y
    cmpl %eax,%edx       # x : y
    jle L9               # if <= goto L9
    movl %edx,%eax       # eax = x
L9:                      # Done:
```

Skipped wh

Beispiel: „Do-While“ Schleife

- C Code

```
int fact_do
   (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Rückwärtssprung setzt Schleife fort
- wird nur ausgeführt, wenn die „while“ Bedingung gilt

Beispiel: „Do-While“ Schleife (cont.)

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Register

%edx x

%eax result

```
_fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg L11              # if > goto loop

  movl %ebp,%esp      # Finish
  popl %ebp           # Finish
  ret                 # Finish
```

Allgemeine „Do-While“ Übersetzung

C Code

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

Goto Version

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

- beliebige Folge von Anweisungen als Schleifenkörper
- Abbruchbedingung basiert auf Integer-Wert
  - $\neq 0$ entspricht wahr: nächste Iteration
  - $= 0$ entspricht falsch: Schleife verlassen

„While“ Übersetzung

C Code

```
while (Test)
  Body
```

Do-While Version

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

Goto Version

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

„For“ Übersetzung

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:
```

Mehrfachverzweigungen „Switch“

- 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

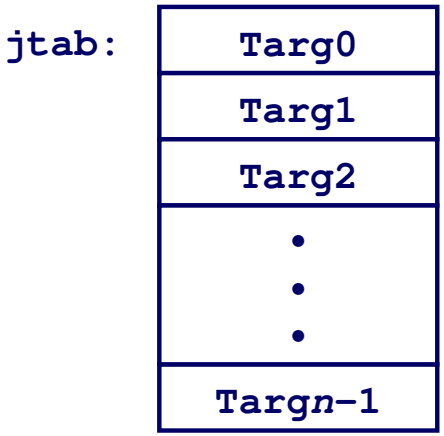
```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Sprungtabelle

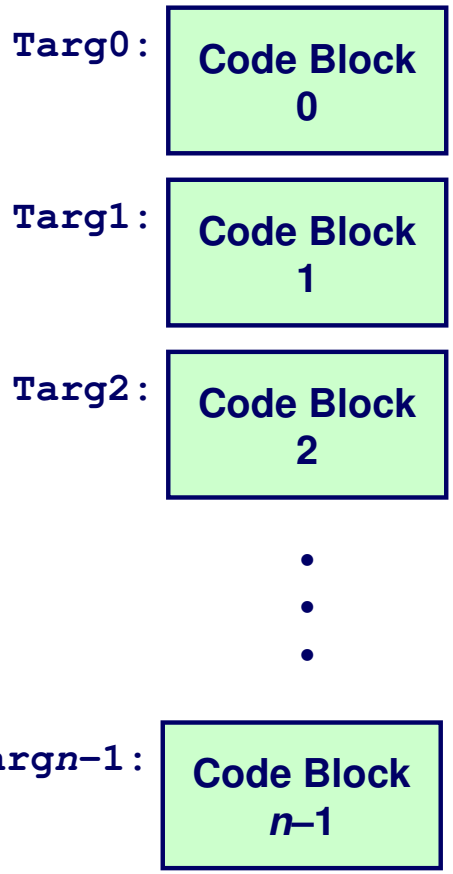
Switch Form

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

Jump Table



Jump Targets



Approx. Translation

```
target = JTab[op];
goto *target;
```

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

Beispiel: „Switch“

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

```
ADD   0
MULT  1
MINUS 2
DIV   3
MOD   4
BAD   5
```

Setup:

```
unparse_symbol:
   pushl %ebp          # Setup
   movl %esp,%ebp      # Setup
   movl 8(%ebp),%eax   # eax = op
   cmpl $5,%eax        # Compare op : 5
   ja .L49             # If > goto done
   jmp *.L57(,%eax,4) # goto Table[op]
```

Beispiel: „Switch“ (cont.)

- Compiler erzeugt Code für jeden case Zweig
- je ein Label am Start der Zweige, `.L51` .. `.L56`

- Tabellenstruktur
  - jedes Ziel benötigt 4 Bytes (32-bit Code)
  - Basisadresse bei `.L57`

- Sprünge
  - `jmp *.L57(,%eax, 4)`
    - Sprungtabelle ist mit Label `.L57` gekennzeichnet
    - Register %eax speichert *op*
    - Skalierungsfaktor 4 für Tabellenoffset
    - Sprungziel: effektive Adresse `.L57` + *op*×4
  - `jmp .L49` markiert das Ende der switch-Anweisung

Beispiel: „Switch“ (cont.)

Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

```
ADD    0
MULT   1
MINUS  2
DIV    3
MOD    4
BAD    5
```

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

Sprungtabelle aus Binärcode Extrahieren

```
Contents of section .rodata:
 8048bc0 30870408 37870408 40870408 47870408  0...7...@...G...
 8048bd0 50870408 57870408 46616374 28256429  P...W...Fact(%d)
 8048be0 203d2025 6c640a00 43686172 203d2025   = %ld..Char = %
 …
```

- im read-only Datensegment gespeichert (`.rodata`)
  - dort liegen konstante Werte des Codes
- kann mit `objdump` untersucht werden
  `objdump code-examples -s --section=.rodata`
  - zeigt alles im angegebenen Segment
  - schwer zu lesen (!)
  - Einträge der Sprungtabelle in umgekehrter Byte-Anordnung
    z.B: 30870408 ist eigentlich 0x08048730

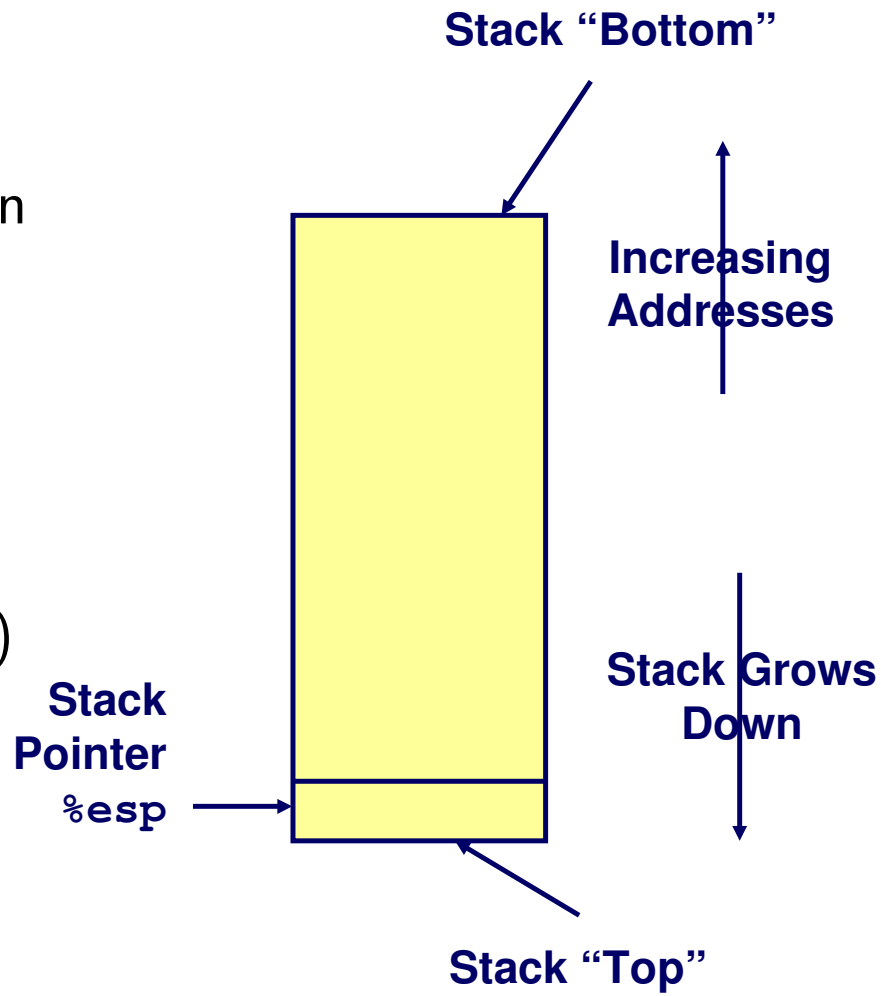
Zusammenfassung: x86 Assembler

- Primitive Operationen und Adressierung
- C Kontrollstrukturen
  - „if-then-else“
  - „do-while“, „while“, „for“
  - „switch“
- Assembler Kontrollstrukturen
  - „Jump“
  - „Conditional Jump“
- Compiler
  - erzeugt Assembler Code für komplexere C Kontrollstrukturen
  - alle Schleifen in „do-while“ / „goto“ Form konvertieren
  - Sprungtabellen für „switch“ Mehrfachverzweigungen

x86 Stack (Kellerspeicher)

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

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



Stack (Kellerspeicher)

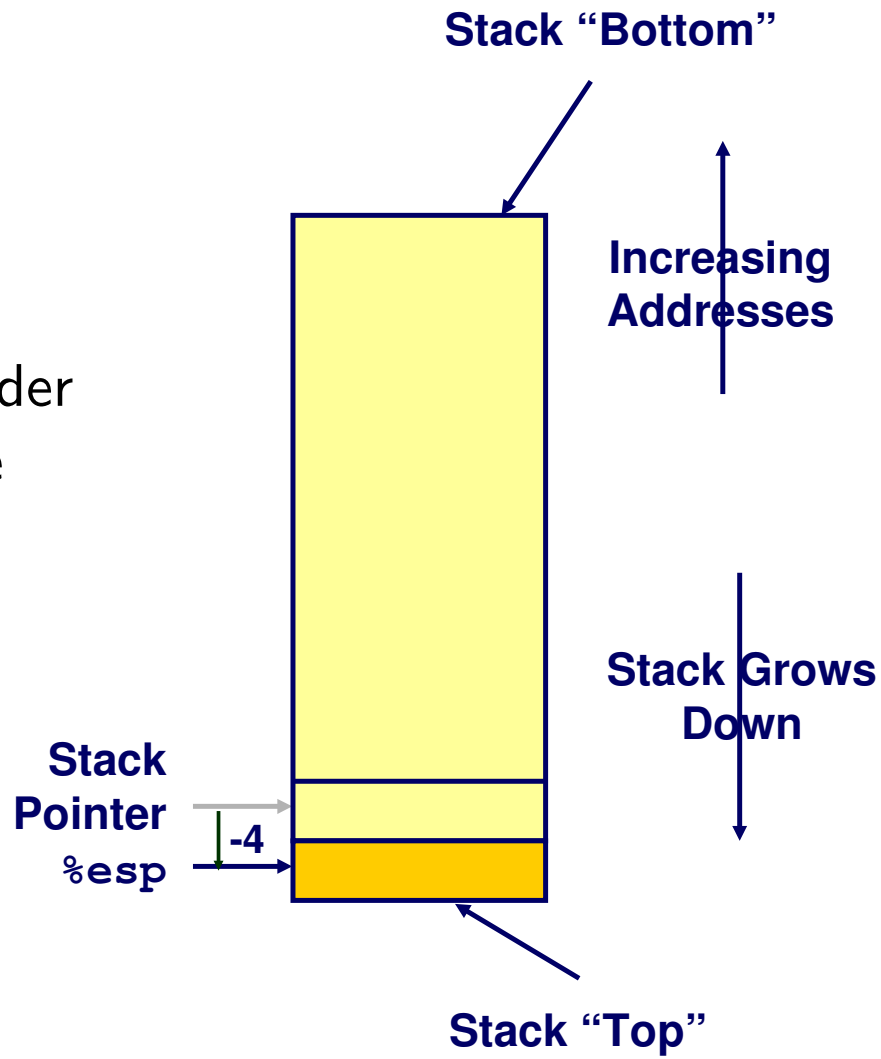
- Implementierung von Funktionen/Prozeduren
  - Speicherplatz für Aufruf-Parameter
  - Speicherplatz für lokale Variablen
  - Rückgabe der Funktionswerte
  - auch für rekursive Funktionen (!)

- mehrere Varianten/Konventionen
  - Parameterübergabe in Registern
  - caller-save
  - callee-save
  - Kombinationen davon
  - Aufruf einer Funktion muss deren Konvention berücksichtigen

Stack: Push

`pushl` ⟨*src*⟩

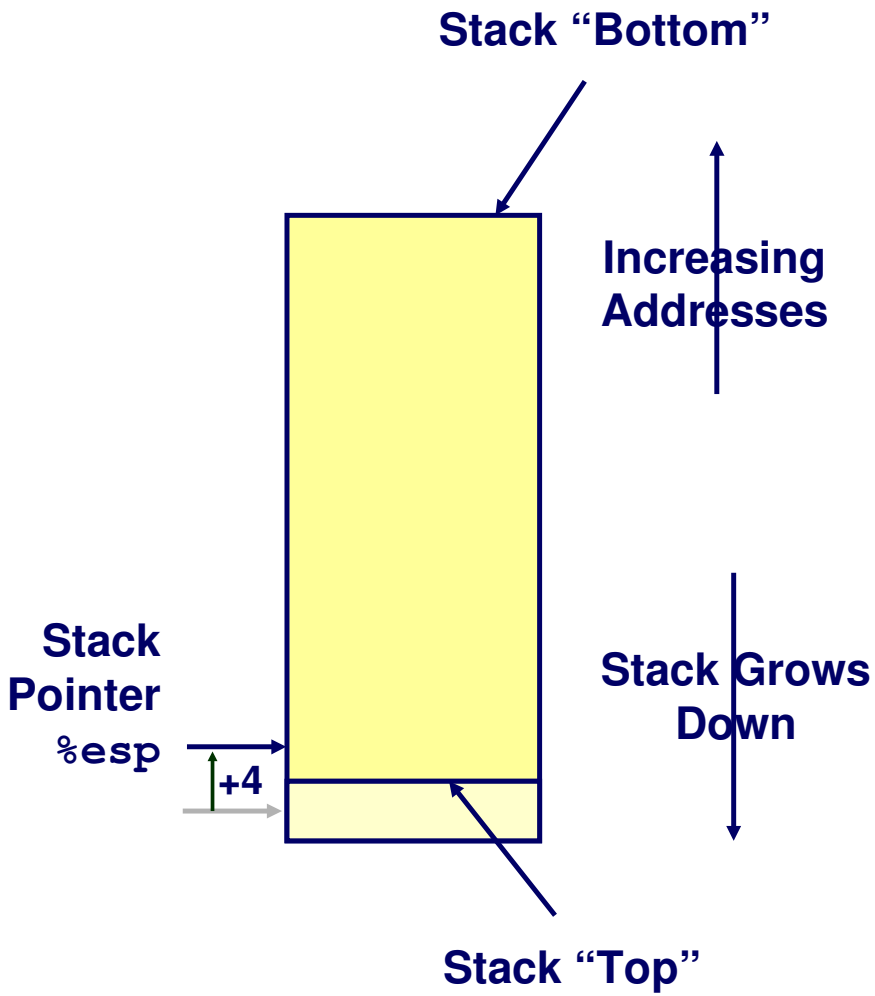
- holt Operanden aus ⟨*src*⟩
- dekrementiert %esp um 4
- speichert den Operanden unter der von %esp vorgegebenen Adresse



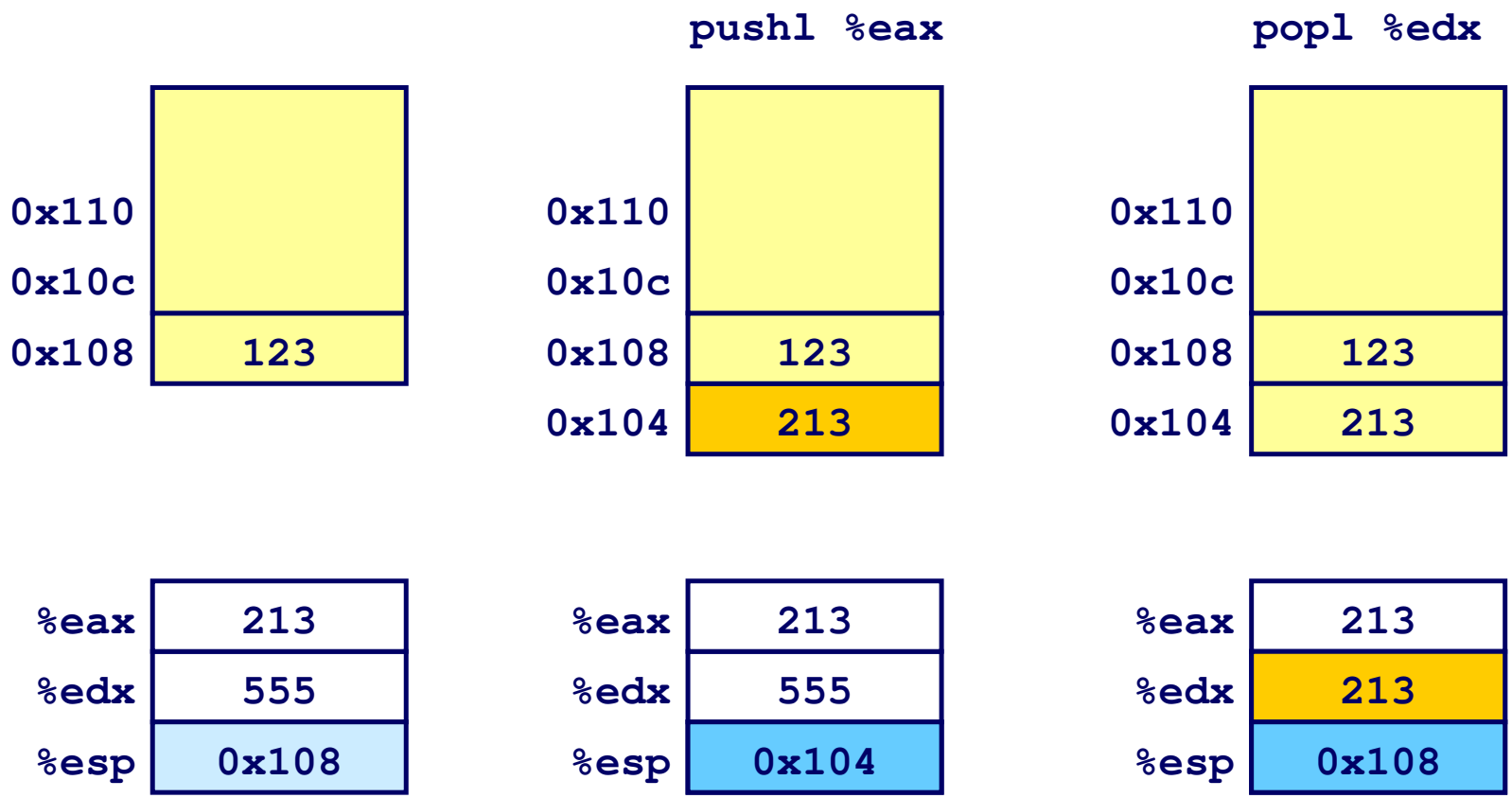
Stack: Pop

`popl` ⟨*dst*⟩

- liest den Operanden unter der von %esp vorgegebenen Adresse
- inkrementiert %esp um 4
- schreibt gelesenen Wert in ⟨*dst*⟩



Beispiele: Stack-Operationen



x86: Funktions-/Prozeduraufruf

- x86 ist CISC: spezielle Maschinenbefehle für Funktionsaufruf
  - `call` zum Aufruf einer Funktion
  - `ret` zum Rücksprung aus der Funktion
  - beide Funktionen ähnlich `jmp`: EIP wird modizfiziert
  - Stack wird zur Parameterübergabe verwendet

- zwei Register mit Spezialaufgaben
  - `%ESP`: „stack-pointer“: Speicheradresse des top-of-stack
  - `%EBP`: „base-pointer“: Speicheradresse der aktuellen Funktion

x86: Funktions/-Prozeduraufruf

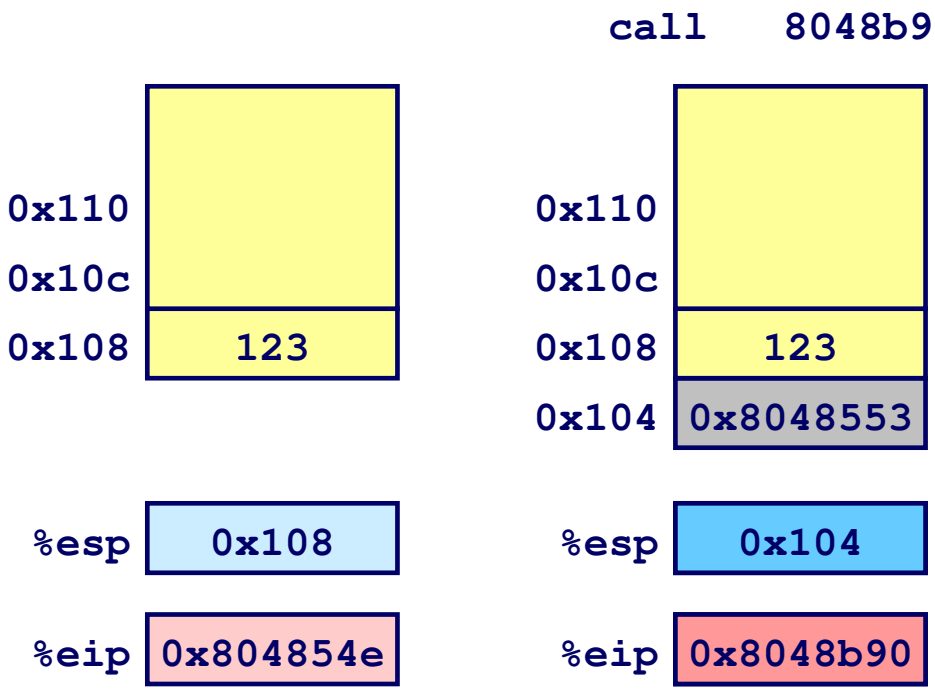
- Prozeduraufruf: `call` ⟨*label*⟩
  - Rücksprungadresse auf Stack („Push“)
  - Sprung zu ⟨*label*⟩
- Wert der Rücksprungadresse
  - Adresse der auf den `call` folgenden Anweisung
  - Beispiel:

```
804854e:   e8 3d 06 00 00   ;call 8048b90
8048553:   50               ;pushl %eax
⟨main⟩     ...              ;...
8048b90:                    ;Prozedureinsprung
⟨proc⟩     ...              ;...
...        ret              ;Rücksprung
```

  - Rücksprungadresse `0x8048553`
- Rücksprung `ret`
  - Rücksprungadresse vom Stack („Pop“)
  - Sprung zu dieser Adresse

x86: call Prozeduraufruf

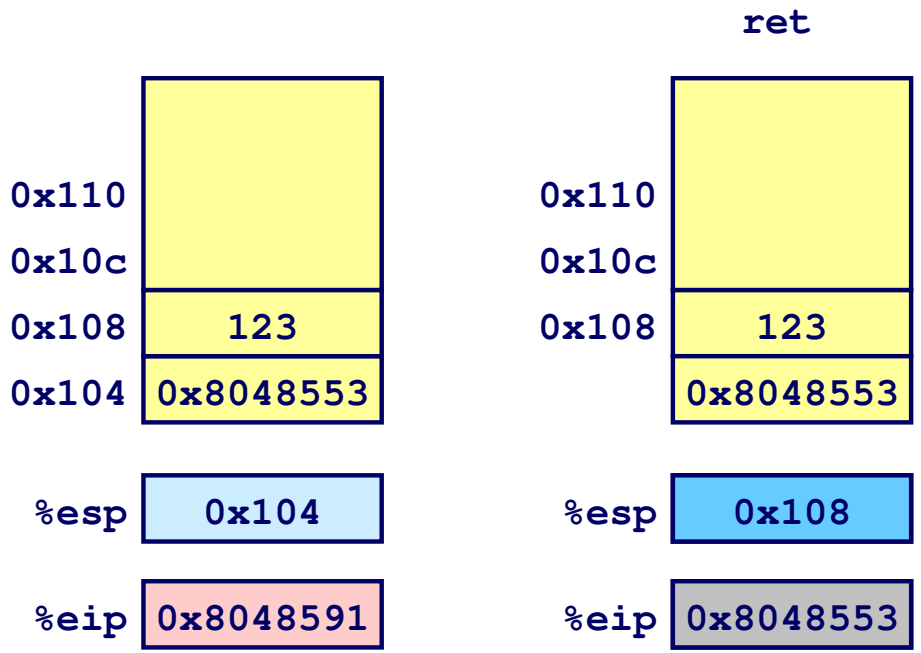
```
804854e:  e8 3d 06 00 00      call   8048b90 <main>
8048553:  50                  pushl  %eax
```



%eip is program counter

x86: ret Rücksprung

```
8048591:   c3                  ret
```



%eip is program counter

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
    - 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
  - Aufruf-Parameter der Funktion/Prozedur
  - Rücksprungadresse
  - lokale Variablen
  - temporäre Daten
- Verwaltung
  - beim Aufruf wird Speicherbereich zugeteilt „setup“ Code
  - beim Return wird Speicherbereich freigegeben „finish“ Code
- Adressenverweise („Pointer“)
  - Stackpointer %esp gibt das obere Ende des Stacks an
  - Framepointer %ebp gibt den Anfang des aktuellen Frame an

Beispiel: Stack-Frame

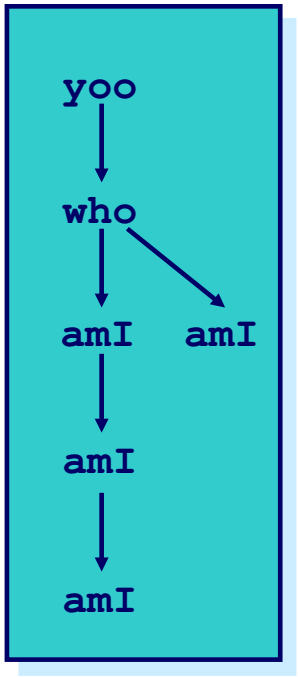
Code Structure

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

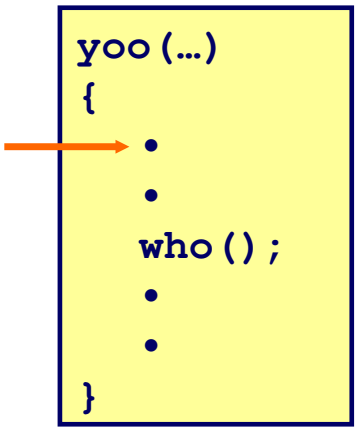
```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

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

Call Chain

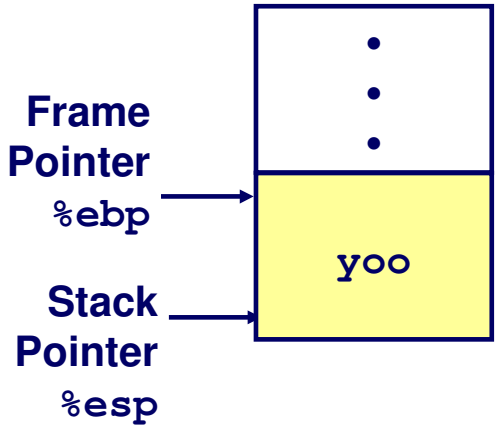


Beispiel: Stack-Frame (cont.)

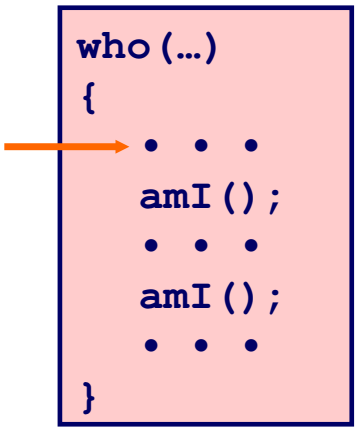


Call Chain

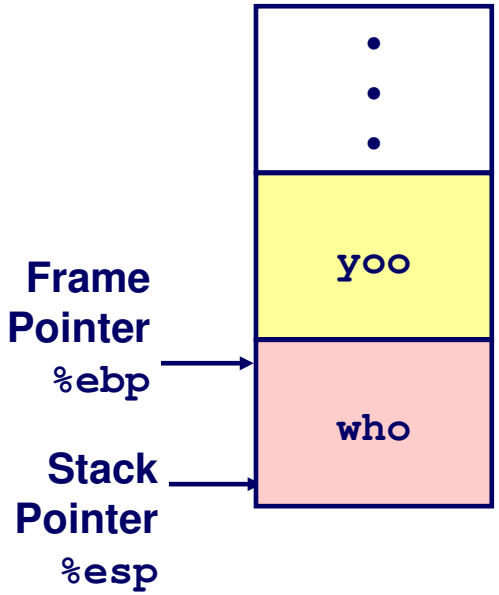
yoo



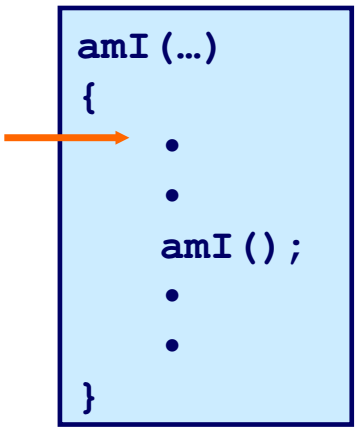
Beispiel: Stack-Frame (cont.)



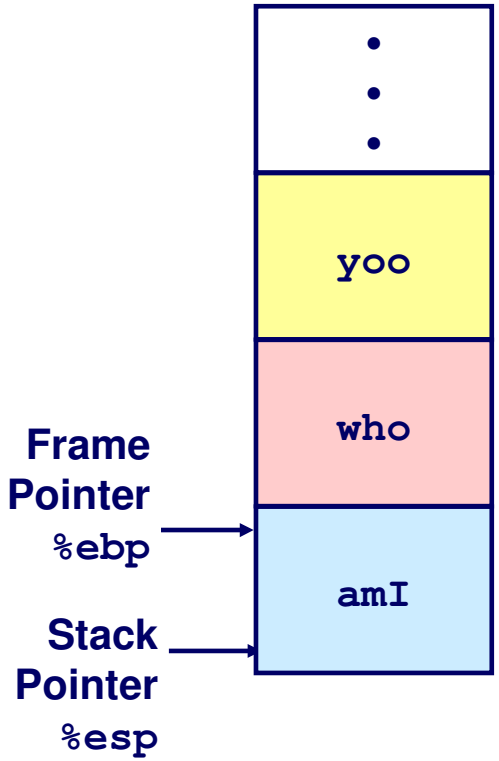
Call Chain



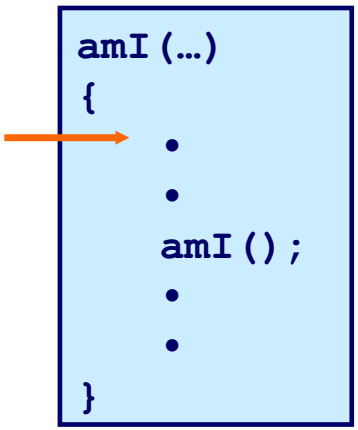
Beispiel: Stack-Frame (cont.)

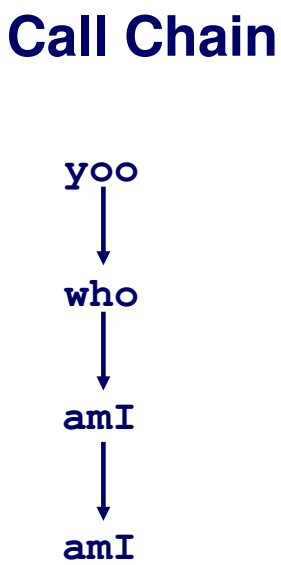


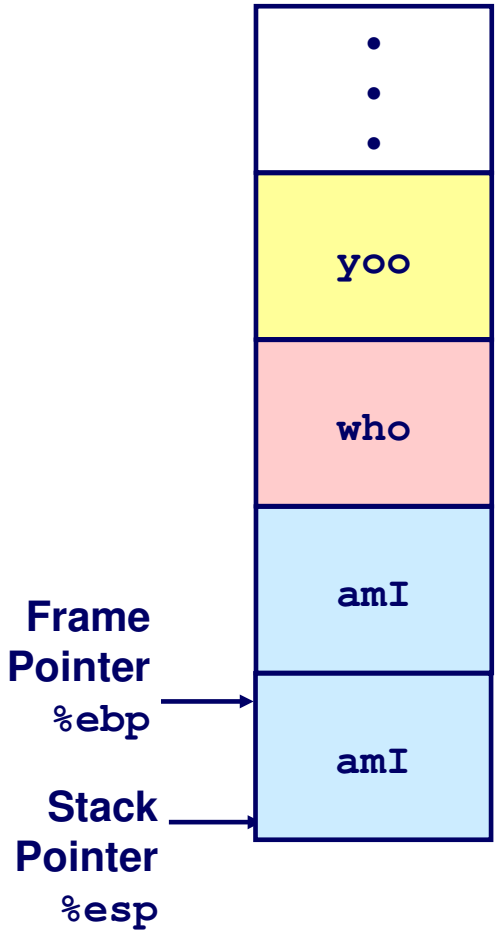
Call Chain



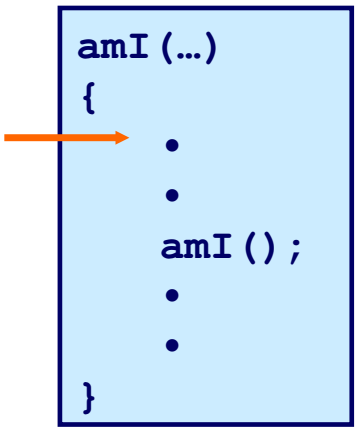
Beispiel: Stack-Frame (cont.)

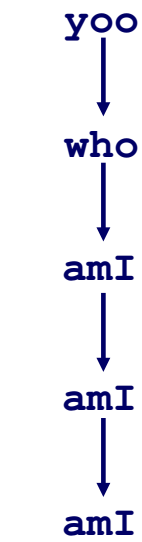


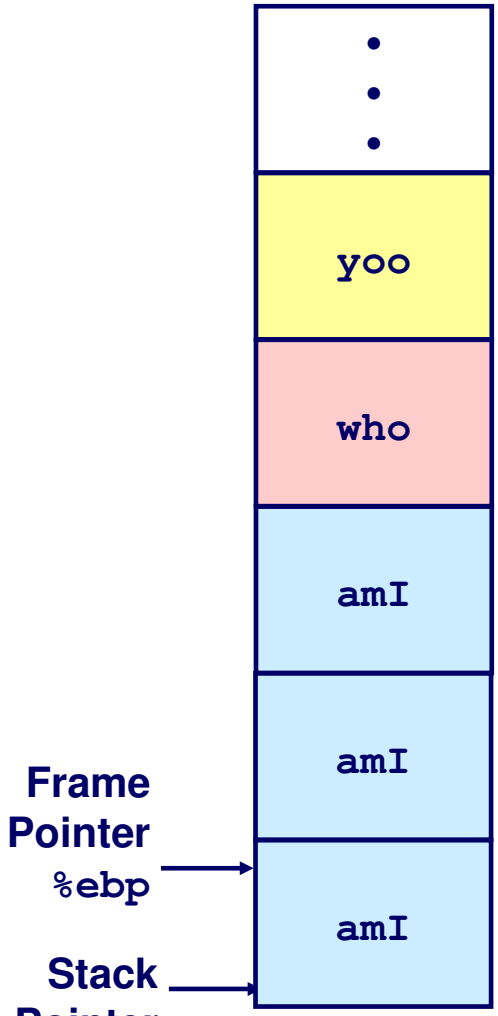




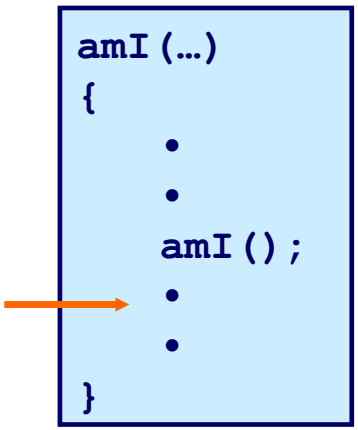
Beispiel: Stack-Frame (cont.)

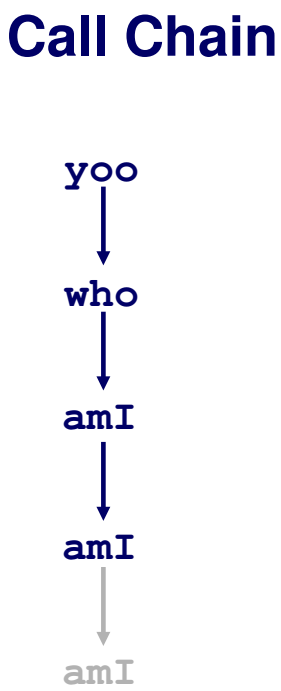


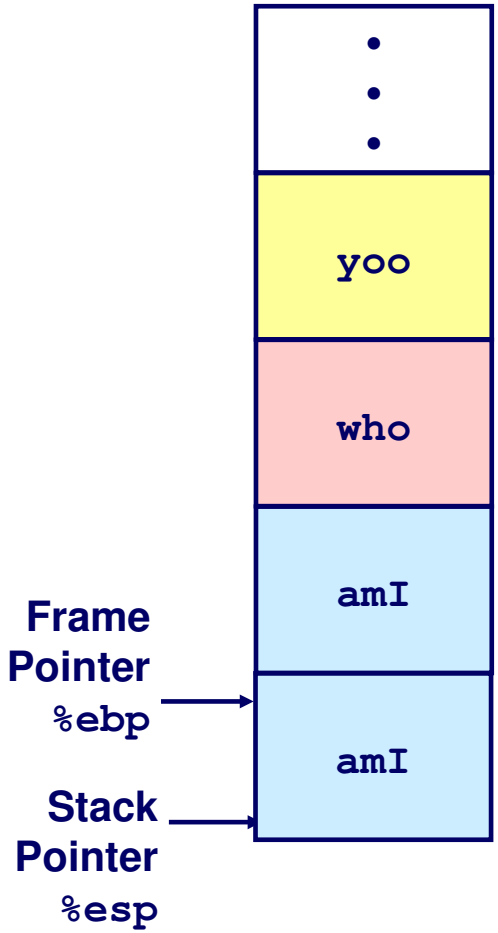




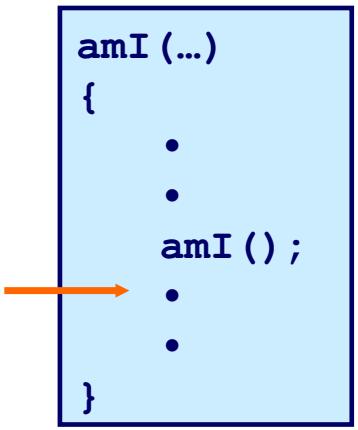
Beispiel: Stack-Frame (cont.)

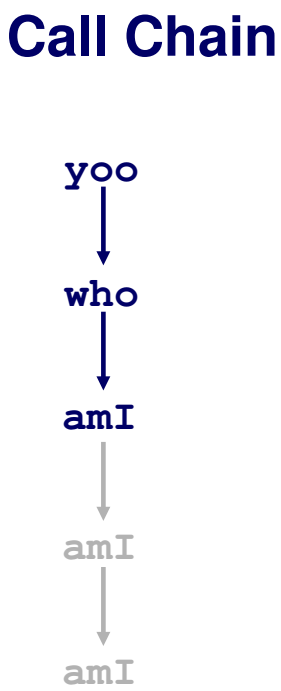


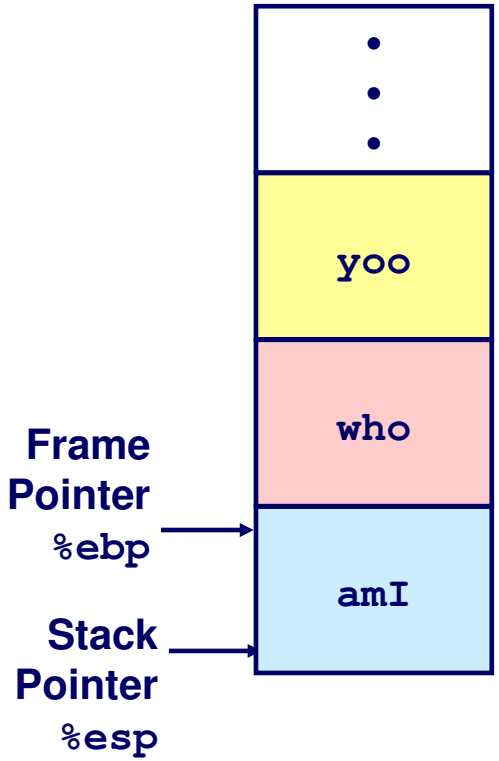




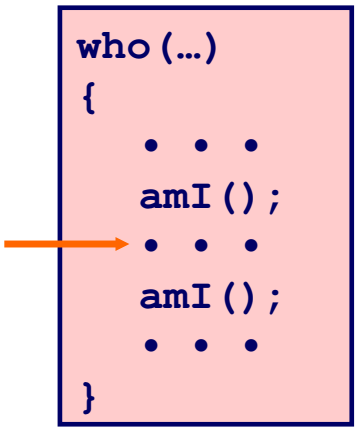
Beispiel: Stack-Frame (cont.)

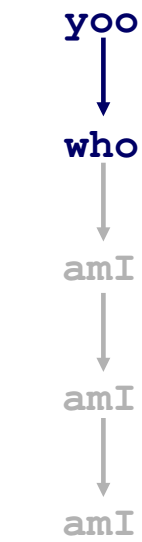


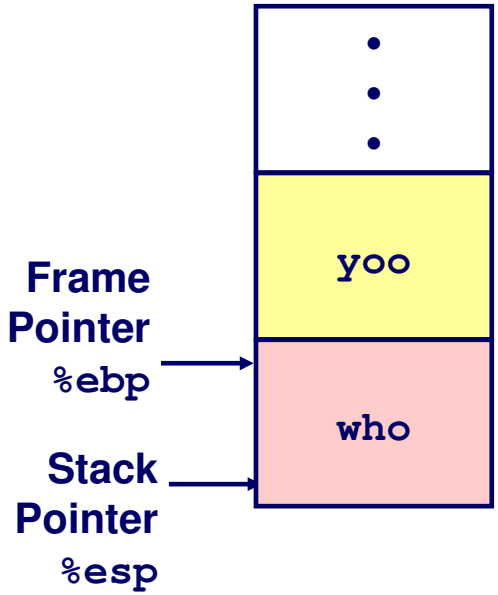




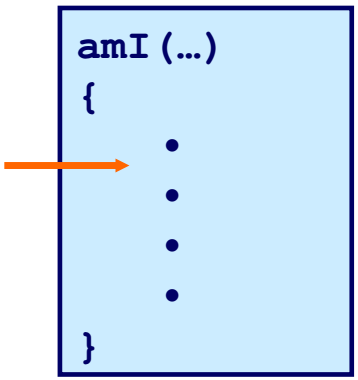
Beispiel: Stack-Frame (cont.)

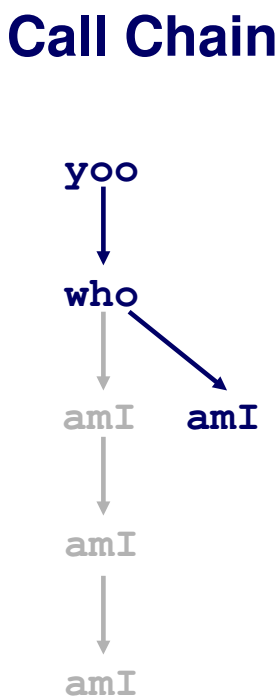


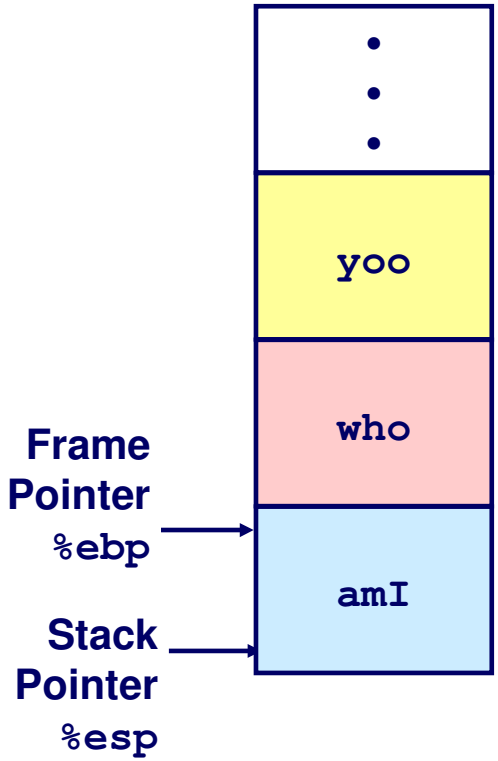




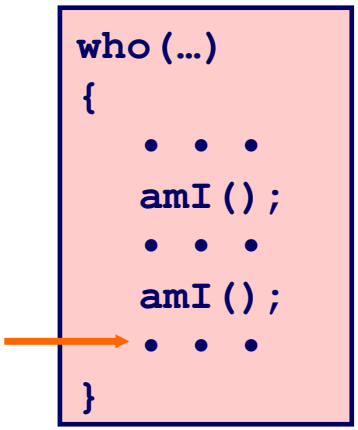
Beispiel: Stack-Frame (cont.)

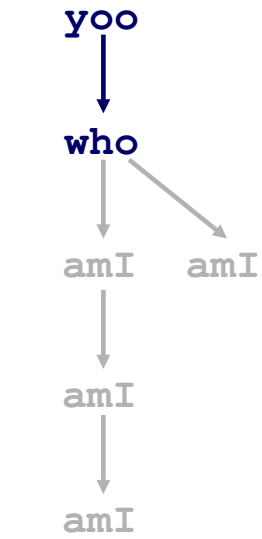


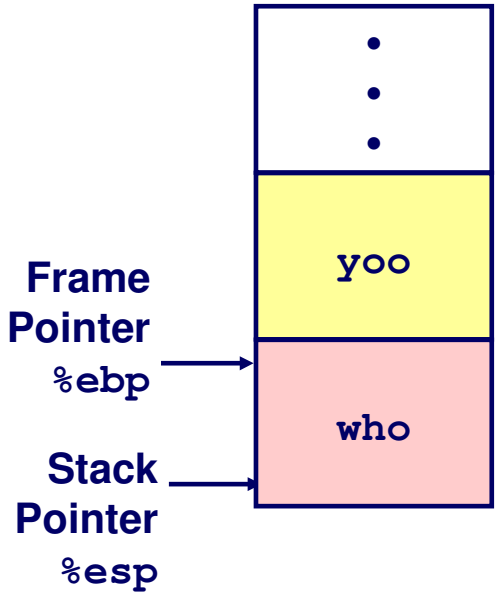




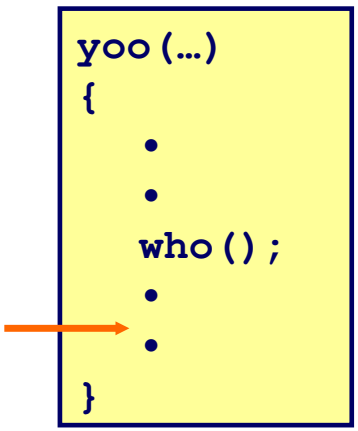
Beispiel: Stack-Frame (cont.)

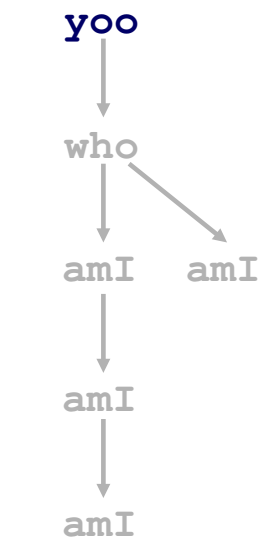


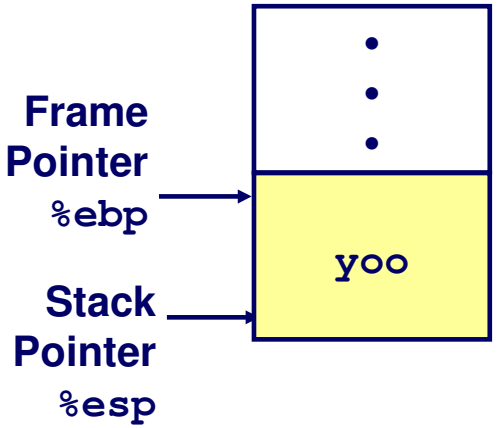




Beispiel: Stack-Frame (cont.)



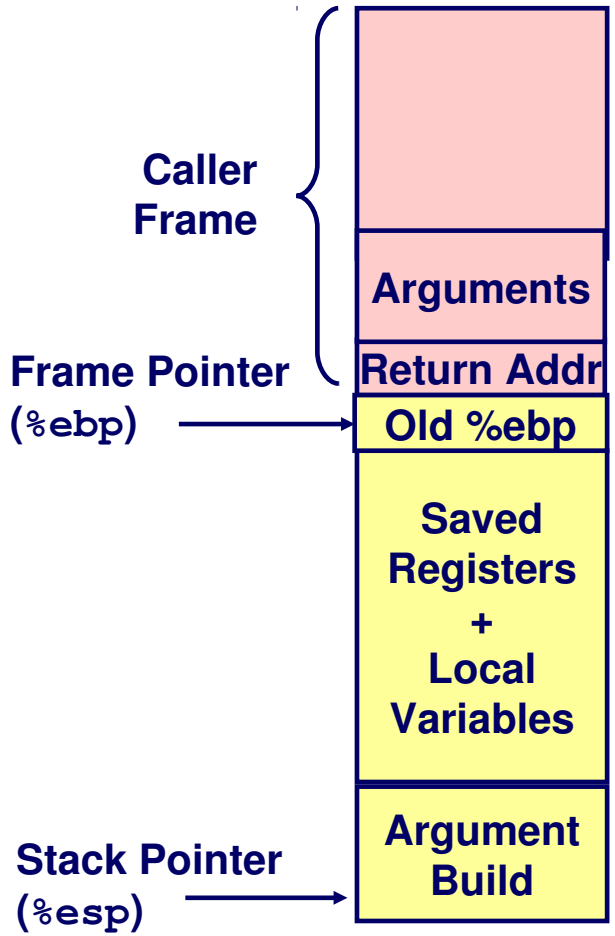




x86/Linux Stack-Frame

aktueller Stack-Frame

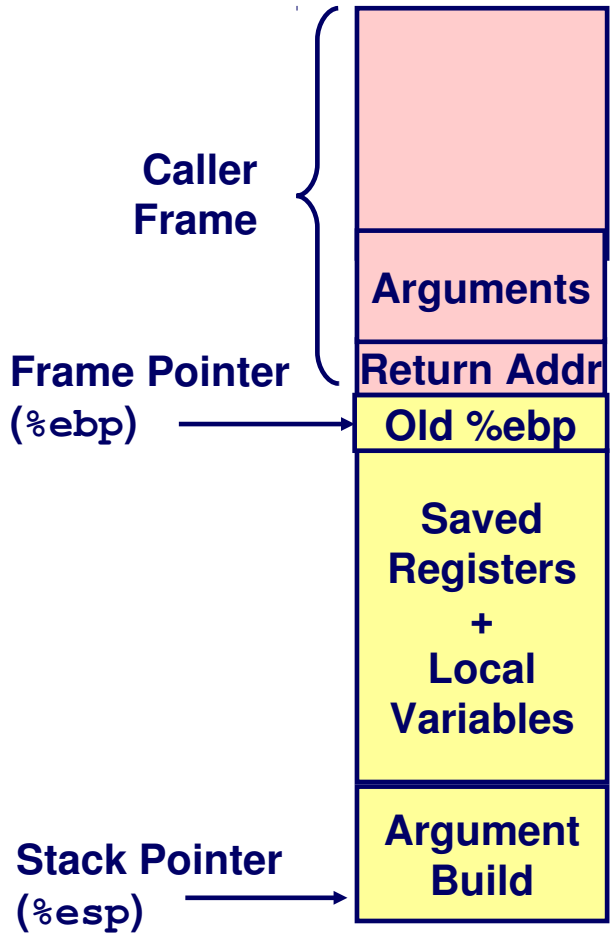
- von oben nach unten organisiert „Top“. . .„Bottom“
- Parameter für weitere Funktion die aufgerufen wird `call`
- lokale Variablen
  - wenn sie nicht in Registern gehalten werden können
- gespeicherter Registerkontext
- Zeiger auf vorherigen Frame



x86/Linux Stack-Frame (cont.)

„Caller“ Stack-Frame

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



Register-Verwendung: Konventionen

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

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

- kann `who` Register für vorübergehende Speicherung benutzen?
  - Inhalt von %edx wird von `who` überschrieben

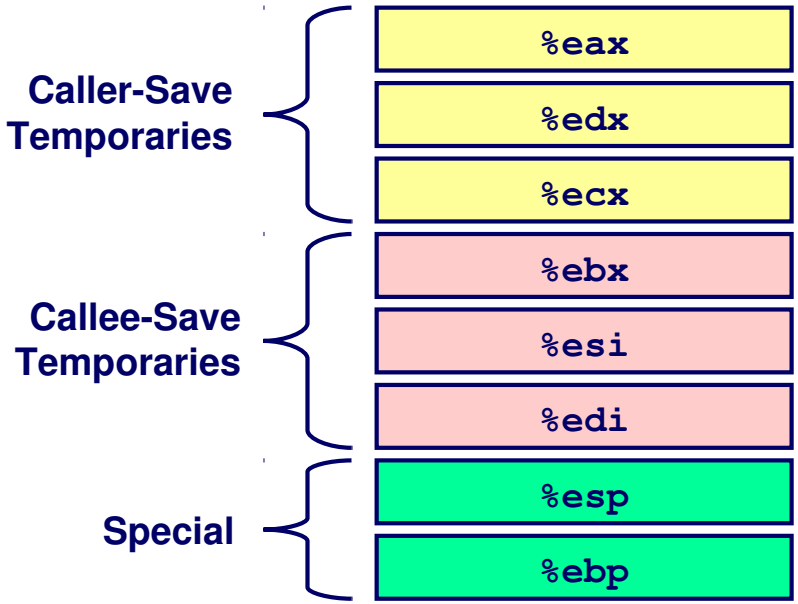
⇒ zwei mögliche Konventionen

- „Caller-Save“
  `yoo` speichert in seinen Frame vor Prozeduraufruf
- „Callee-Save“
  `who` speichert in seinen Frame vor Benutzung

x86/Linux Register Verwendung

Integer Register

- zwei werden speziell verwendet
  - %ebp, %esp
- „Callee-Save“ Register
  - %ebx, %esi, %edi
  - alte Werte werden vor Verwendung auf dem Stack gesichert
- „Caller-Save“ Register
  - %eax, %edx, %ecx
  - “Caller” sichert diese Register
- Register %eax speichert auch den zurückgelieferten Wert



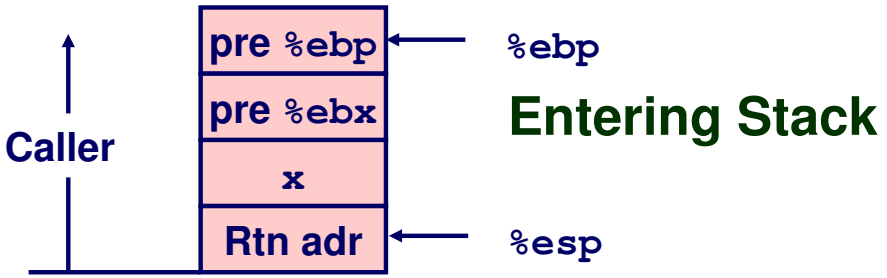
Beispiel: Rekursive Fakultät

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

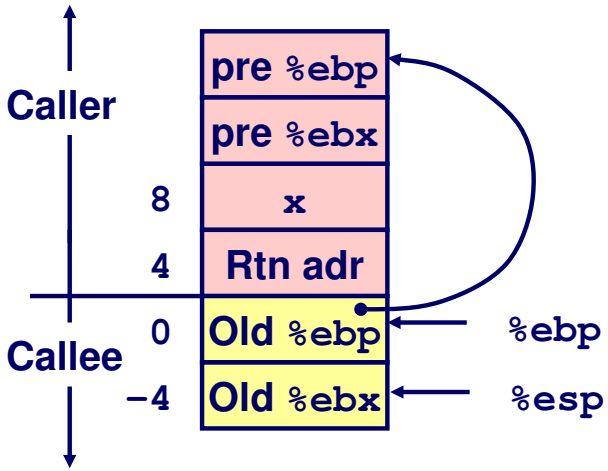
- %eax
  - benutzt ohne vorheriges Speichern
- %ebx
  - am Anfang speichern
  - am Ende zurückschreiben

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Beispiel: `rfact` – Stack „Setup“




```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```



Beispiel: `rfact` – Rekursiver Aufruf

Recursion

```
  movl 8(%ebp),%ebx    # ebx = x
  cmpl $1,%ebx         # Compare x : 1
  jle .L78             # If <= goto Term
  leal -1(%ebx),%eax   # eax = x-1
  pushl %eax           # Push x-1
  call rfact           # rfact(x-1)
  imull %ebx,%eax      # rval * x
  jmp .L79             # Goto done
.L78:                # Term:
  movl $1,%eax         # return val = 1
.L79:                # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

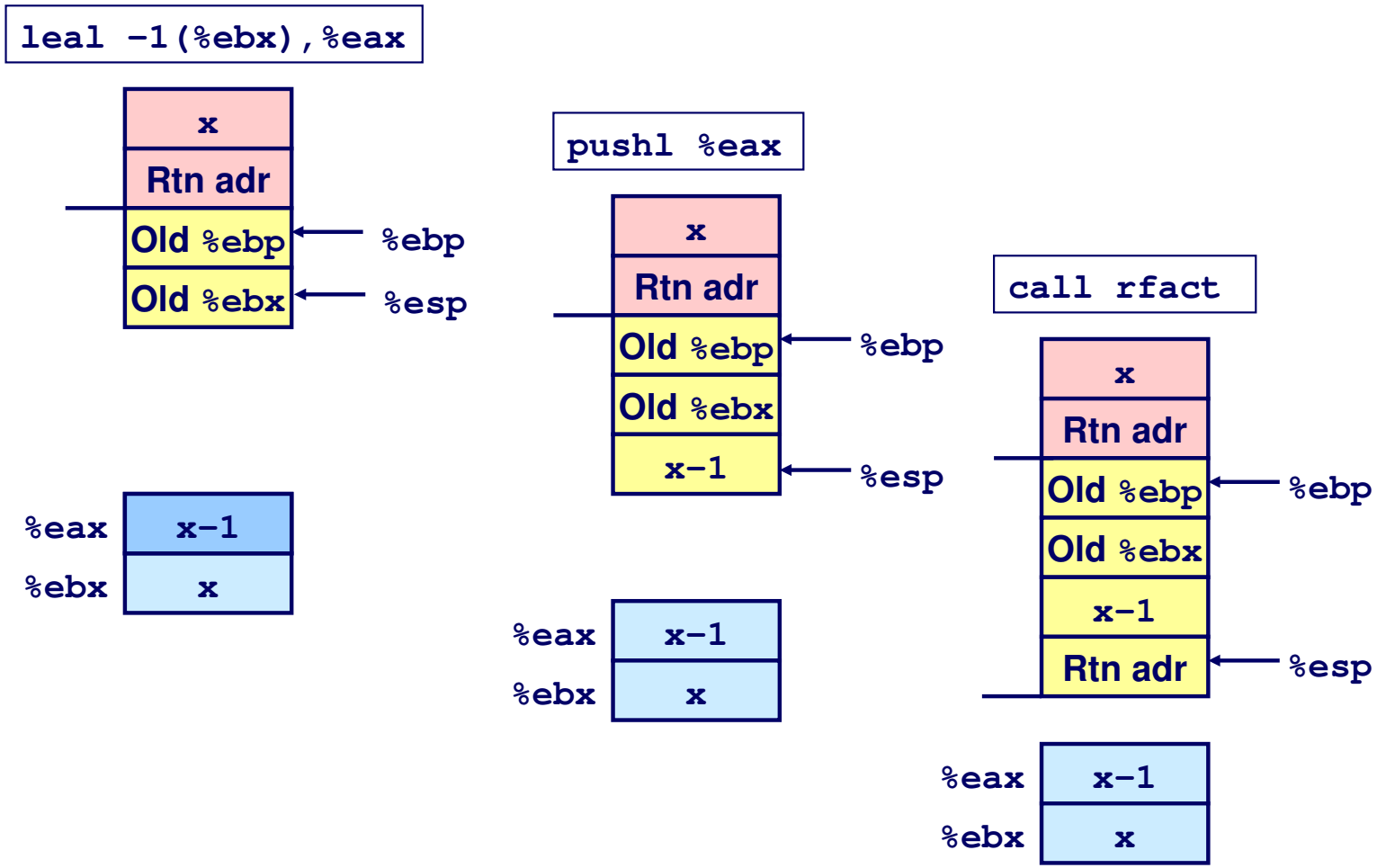
Registers

`%ebx` **Stored value of x**

`%eax`

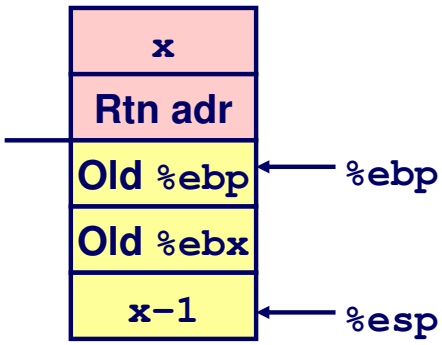
- **Temporary value of** `x-1`
- **Returned value from** `rfact(x-1)`
- **Returned value from this call**

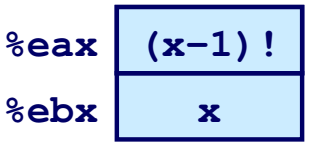
Beispiel: `rfact` – Rekursion



Beispiel: `rfact` – Ergebnisübergabe

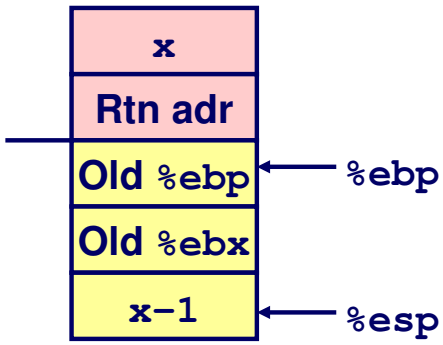
Return from Call

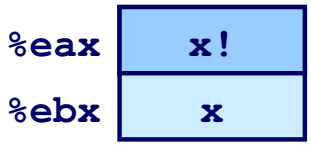




Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

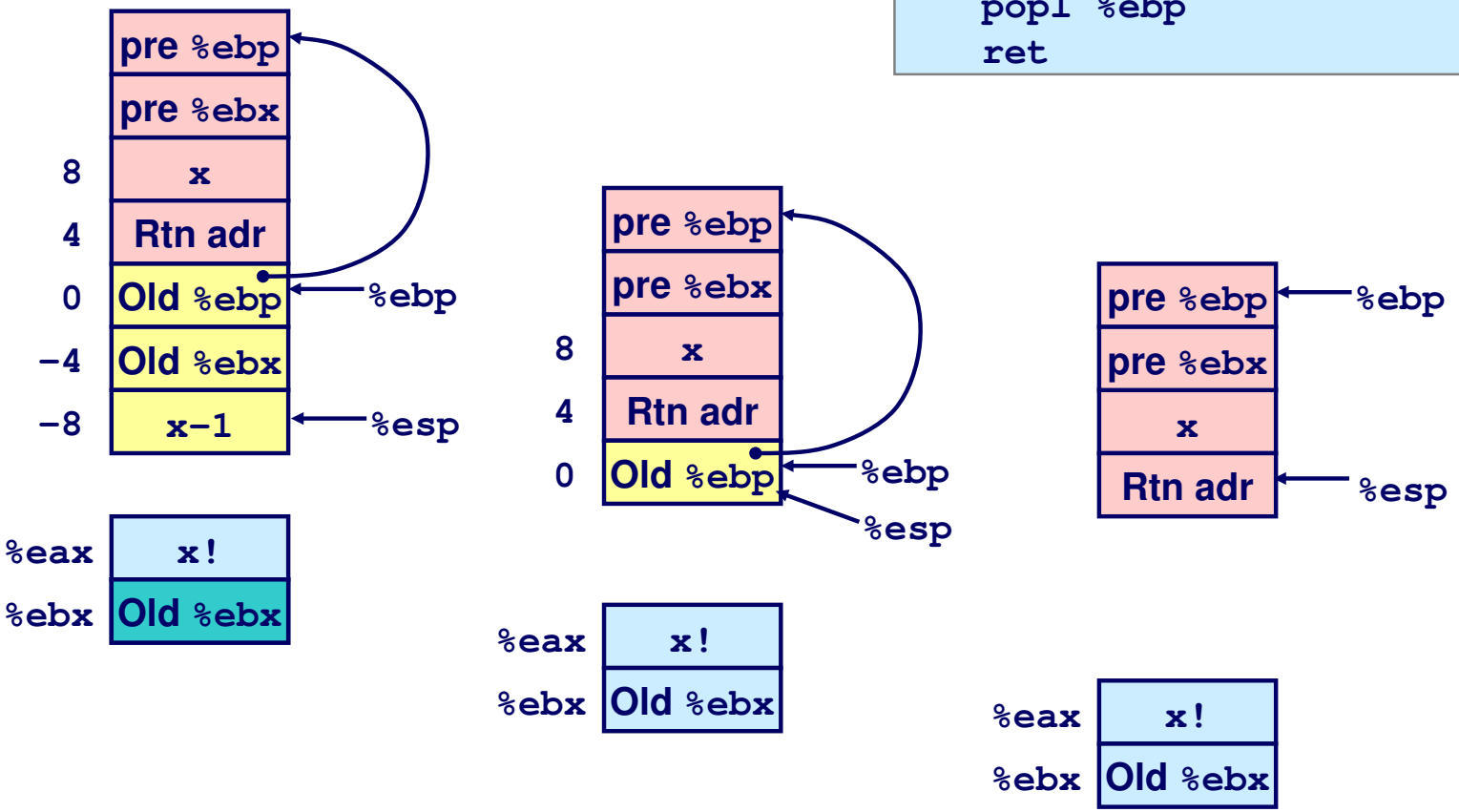
`imull %ebx,%eax`





Beispiel: `rfact` – Stack „Finish“

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```



Zeiger auf Adresse / *call by reference*

- Variable der aufrufenden Funktion soll modifiziert werden

⇒ Adressenverweis (*call by reference*)

- Beispiel: `sfact`

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

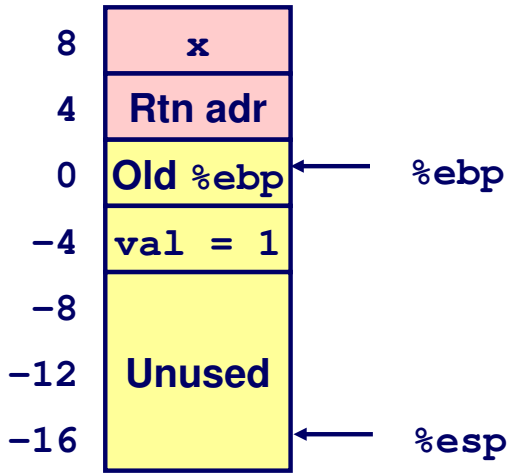
Top-Level Call

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Beispiel: sfact

Initial part of sfact

```
_sfact:
   pushl %ebp          # Save %ebp
   movl %esp,%ebp      # Set %ebp
   subl $16,%esp       # Add 16 bytes
   movl 8(%ebp),%edx   # edx = x
   movl $1,-4(%ebp)    # val = 1
```



- lokale Variable `val` auf Stack speichern
  - Pointer auf `val`
  - berechnen als `-4(%ebp)`
- Push `val` auf Stack
  - zweites Argument
  - `movl $1, -4(%ebp)`

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

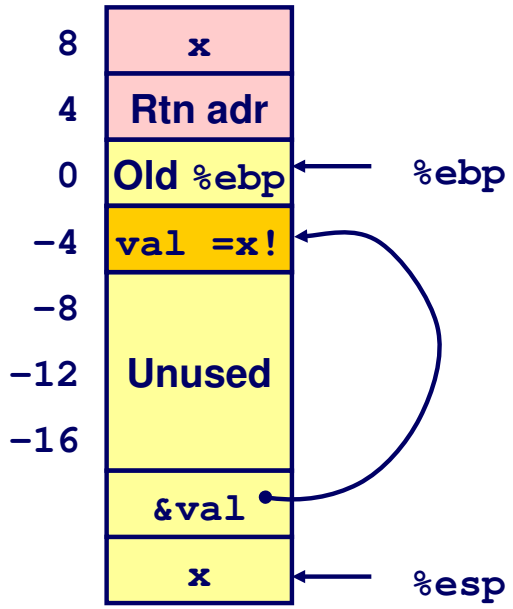
Beispiel: `sfact` – Pointerübergabe bei Aufruf

Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax # Compute &val
pushl %eax         # Push on stack
pushl %edx         # Push x
call s_helper      # call
movl -4(%ebp),%eax # Return val
• • •              # Finish
```

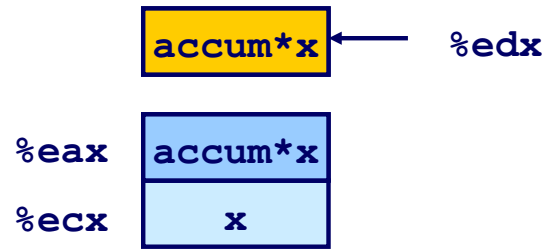
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call



Beispiel: sfact – Benutzung des Pointers

```
void s_helper
  (int x, int *accum)
{
  • • •
    int z = *accum * x;
    *accum = z;
  • • •
}
```




```
  • • •
movl %ecx,%eax     # z = x
imull (%edx),%eax  # z *= *accum
movl %eax,(%edx)   # *accum = z
  • • •
```

- Register %ecx speichert x
- Register %edx mit Zeiger auf accum

Zusammenfassung: Stack

- Stack ermöglicht Funktionsaufrufe und Rekursion
  - lokaler Speicher für jeden Prozeduraufruf („call“)
    - Instanziierungen kommen sich nicht ins Gehege
    - Adressierung lokaler Variablen und Argumente ist relativ zur Stackposition (Framepointer)
  - grundlegendes (Stack-) Prinzip
    - Prozeduren terminieren in umgekehrter Reihenfolge der Aufrufe
- x86 Prozeduren sind Kombination von Anweisungen und Konventionen
  - `call`- und `ret`-Befehle
  - Konventionen zur Registerverwendung
    - „Caller-Save“ / „Callee-Save“
    - `%ebp` und `%esp`
  - festgelegte Organisation des Stack-Frame