

# Unterprogramme

*Aufruf eines Unterprogramms (erster Lösungsansatz):*

```
08:  ...
09:  jmp 20
10:  ...

20:  ...          /* Start Unterprogramm */
...
35:  jmp 10        /* Ende Unterprogramm */
```

*Das Unterprogramm selbst muss in diesem Fall mit jmp 10 zum Hauptprogramm zurückspringen. Problematisch, wenn Unterprogramm von mehreren Stellen aus dem Hauptprogramm heraus aufgerufen werden soll.*

*Man muss sich merken, von wo aus das Unterprogramm aufgerufen wurde bzw. wohin zurückgekehrt werden soll (Return-Address / Rückkehradresse)*

# Unterprogramme

*Aufruf eines Unterprogramms (zweiter Lösungsansatz):*

```
08:  ...
09:  movl $11, %eax
10:  jmp 20
11:  ...
12:  movl $14, %eax
13:  jmp 20
14:  ...
...

20:  ...          /* Start Unterprogramm */
...
35:  jmp %eax      /* Ende Unterprogramm */
```

*Unterprogramm kehrt jetzt zum jeweiligen Aufrufort zurück. Dennoch problematisch: verschachtelte Unterprogrammaufrufe nicht möglich (jedes Register kann nur maximal eine Rücksprungadresse speichern, Anzahl der Register beschränkt)*

# Unterprogramme

*Aufruf eines Unterprogramms:*

```
08: ...
09: movl $11, (%esp)+
10: jmp 20
11: ...

20: ...          /* Start Unterprogramm "func" */
...
35: jmp -(%esp)
```

*%esp (Stack-Pointer) muss zu Programmbeginn die Adresse eines genügend großen Speicherbereiches (Stack) enthalten.*

*Funktioniert für beliebig verschachtelte Unterprogramme (auch Rekursion!).*

# Unterprogramme (Stack)

*Vier verschiedene Möglichkeiten für die Stack-Verwaltung:*

*Richtung des Stacks:*

- Stack wird von den kleinen Adressen hin zu den großen Adressen beschrieben (1)
- Stack wird von den großen Adressen hin zu den kleinen Adressen beschrieben (2)

*Stack-Pointer:*

- Stack-Pointer zeigt auf das zuletzt abgelegte Datum (A)
- Stack-Pointer zeigt auf den nächsten freien Speicherplatz (B)

*Dementsprechend sind zu verwenden:*

„push“	„pop“	
mov ..., +(%esp)	mov (%esp)-, ...	(1A)
mov ..., (%esp)+	mov -(%esp), ...	(1B)
mov ..., -(%esp)	mov (%esp)+, ...	(2A)
mov ..., (%esp)-	mov +(%esp), ...	(2B)

# Unterprogramme (Stack)

## *Hinweise:*

- Stack des i80x86 und m68x05 wächst von den hohen Adressen hin zu den niedrigen
- Stack-Pointer des i80x86 zeigt auf das zuletzt abgelegte Datum.
- Stack-Pointer des m68x05 zeigt auf den nächsten freien Platz auf dem Stack.

## *andere Schreibweise i80x86:*

```
movl $11, -(%esp); jmp 20      => call 20
jmp (%esp)+                    => ret
```

## *andere Schreibweise m68x05:*

```
movl $11, (%sp)-; jmp 20      => jsr 20
jmp +(%sp)                     => rts
```

# Unterprogramme (Parameter / Return-Wert)

## *Verschiedene Möglichkeiten:*

### *Übergabe der Parameter / des Return-Werts über*

- Register
  - Anzahl beschränkt
  - nicht geeignet für rekursive Unterprogramme
- festgelegte Speicherzellen
  - langsam
  - nicht geeignet für rekursive Unterprogramme
- Stack
  - langsam

**=> Übergabe Parameter über Stack, Rückgabe Return-Wert über Register (z.T. Mischformen)**

## Unterprogramme (Parameter / Return-Wert)

*Beispiel: Übergabe der Parameter und des Return-Wertes über die Register:*

Hauptprogramm:

```
...
movl y, %eax
movl z, %ebx
call subtract          x = subtract(y, z);
movl %eax, x
...
```

Unterprogramm subtract(a, b):

```
subtract:
    subl %ebx, %eax    return a - b;
    ret
```

## Unterprogramme (Parameter / Return-Wert)

*Beispiel: Übergabe der Parameter auf dem Stack und des Return-Wertes über ein Register:*

Hauptprogramm:

```
movl ..., %esp
...
movl z, -(%esp)
movl y, -(%esp)
call subtract          x = subtract(y, z);
addl $8, %esp
movl %eax, x
...
```

Unterprogramm subtract(a, b):

```
subtract:
    movl 4(%esp), %eax
    subl 8(%esp), %eax    return a - b;
    ret
```

# Unterprogramme (lokale Variablen)

*Viele Unterprogramme verwenden lokale (temporäre) Variablen (z.B. auch für Berechnung von Ausdrücken).*

## *Speicherung dieser Variablen in Registern*

- schneller Zugriff, kurze Befehle (einfache Adressierungsart), wenige Register mit (sehr) beschränkter Speicherkapazität, keine Rekursion (direkt) möglich

## *... im Speicher an festgelegten Adressen*

- i.a. genügend Speicherkapazität, langsamer Zugriff, keine Rekursion (direkt) möglich, alter Wert beim wiederholten Aufruf des Unterprogramms noch intakt

## *... oder auf dem Stack*

- i.a. genügend Speicherkapazität, kurze Befehle (einfache Adressierungsart), Rekursion direkt möglich, langsamer Zugriff

*=> Mischformen aus allen Speicherorten*

# Unterprogramme (lokale Variablen)

## *Register-Variablen*

- für die am häufigsten verwendeten Variablen (z.B. Schleifen-Zähler und temporäre Variablen zur Ausdrucksberechnung)

## *Variablen im Speicher an festgelegten Adressen*

- für Werte, die beim Wiederholten Aufruf des Unterprogramms intakt sein müssen (static-Werte)

## *Variablen auf dem Stack*

- für seltener verwendete Variablen oder große Variablen (z.B. Records oder Arrays)
- Zwischenspeicher für Register-Variablen (z.B. für Rekursion)

## Unterprogramme (Stack-Variablen)

### *Benutzung von Variablen auf dem Stack*

*(vorausgesetzt, der Stack-Pointer (%esp) zeigt auf das Ende eines freien Speicherbereiches):*

### *Reservieren von Speicherplatz:*

```
subl $nbytes, %esp
```

### *Benutzung der Variablen (Beispiele):*

```
movl 4(%esp), %eax
addl %eax, 16(%esp)
movb 8(%esp), 9(%esp)
...
```

### *Freigeben des Speicherbereiches:*

```
addl $nbytes, %esp
```

## Unterprogramme (Stack-Variablen)

### *Beispiel (i80x86):*

```
void proc(void) {
    long int a;
    short int b;

    a = ...;
    b = ...;
    ...
}
```

### *ergibt compiliert:*

```
proc: subl $6, %esp
      movl ..., 0(%esp)
      movw ..., 4(%esp)
      ...
      addl $6, %esp
      ret
```

# Unterprogramme (Stack-Frame)

## *Jedes Unterprogramm*

- kann Aufruf-Parameter bekommen,
- verfügt über eine Rückkehr-Adresse,
- benützt u.U. lokale Variablen,
- verwendet Stack für neue Unterprogramm-Aufrufe.

*Dieser Bereich des Stacks wird als Stack-Frame des Unterprogramms bezeichnet.*

# Unterprogramme (Stack-Frame)

## *Beispiel:*

```
void proc(long int x, short int y) {  
    long int z;  
    char c;  
    ...  
    func(z, c);  
}
```

...	:	Stack-Frame aufgerufenes Unterprogramm
Parameter z für func	\	
Parameter c für func		/
Stack-Variable z		
Stack-Variable c		Stack-Frame des Unterprogramms
Return-Adresse		\
Parameter x		
Parameter y	/	
...	:	Stack-Frame aufrufendes Unterprogramm

## Unterprogramme (Frame-Pointer)

*Problem: relative Adressen (bzgl. %esp) der lokalen Stack-Variablen und Unterprogramm-Parameter ändert sich mit jedem push bzw. pop (z.B. bei Unterprogramm-Aufrufen):*

```
long int x;  
...  
proc(x, x);  
...
```

*ergibt compiliert:*

```
subl $4, %esp  
...  
pushl 0(%esp)  
pushl 4(%esp) /* relative Adresse von x hat sich durch pushl geändert! */  
call proc  
addl $8, %esp  
...
```

*=> Zugriff auf Stack-Variablen / Parameter sehr mühsam / fehleranfällig zu programmieren*

## Unterprogramme (Frame-Pointer)

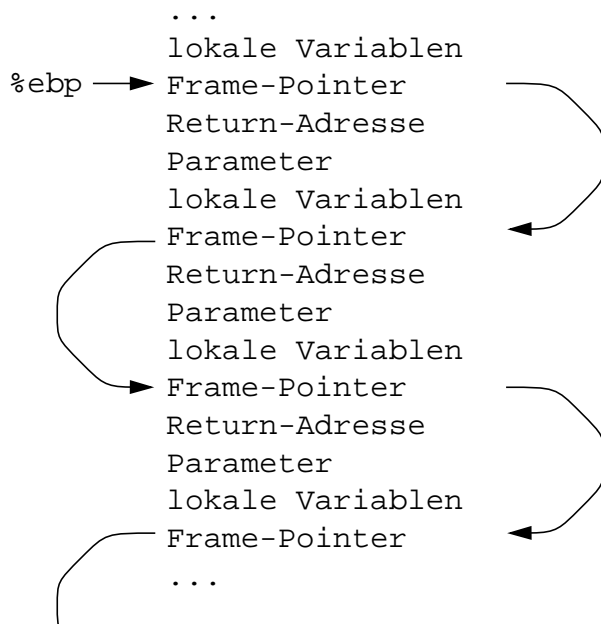
*Extra-Register (Frame-Pointer; i80x86: %ebp) enthält Adresse für Stack-Variablen- und Parameter-Zugriff:*

*Beispiel ergibt compiliert:*

```
pushl %ebp          \  
movl %esp, %ebp    |  enter $4  
subl $4, %esp      /  
...  
pushl -4(%ebp)  
pushl -4(%ebp)     /* %ebp ändert sich durch pushl nicht */  
call proc  
addl $8, %esp  
...  
movl %ebp, %esp    \  
popl %ebp          /  leave  
ret
```

# Unterprogramme (Frame-Pointer)

*Stack-Frames mit Frame-Pointern:*



Durch Verfolgung der Frame-Pointer kann die gesamte Aufruf-Hierarchie der Unterprogramme verfolgt werden (Debugger).