Hochsprachenkonstrukte in Maschinensprache

Moritz Carmesin

18.6.2009

Gliederung

Ein wenig Geschichte . . .

- Ende der 1970: erster Prozessor dieser Architektur: 80**86**
- Intels erster 16-Bit-Prozessor
- Billigere Version 8088 Grundlage der ersten IBM-PCs
- Konnte maximal 1MB (!) Arbeitsspeicher adressieren
- Für Berechnungen mit Dezimalzahlen wurde ein zusätzlicher Koprozessorchip benötigt (8087)
- Nachfolgemodelle 80186 und 80286 für Anwender ohne viele neue Funktionen

Ein wenig Geschichte . . .

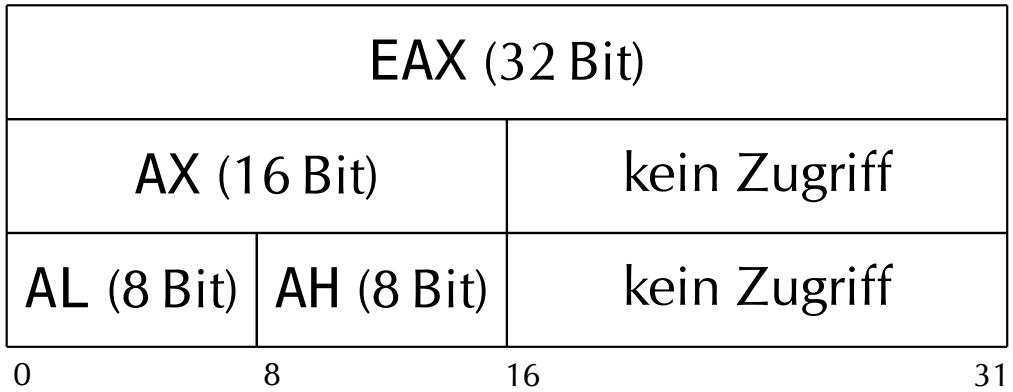
- **1985:** Einführung des 80386
- 32-Bit-Prozessor
- Theoretisch 4 GB Arbeitsspeicher adressierbar
- Hardwareseitige Unterstüztung von Multitasking
- Alle heutigen PC-Prozessoren bauen auf diesem (auch Pentium 4, Core, AMD Phenom etc.)
- x86-Prozessoren immer voll aufwärtskompatibel, d. h. auch Programme für den 8086 laufen theoretisch noch auf einem Intel Core i7 Prozessor
- Mit neuen Prozessor-Generationen neue zusätzliche Befehle

Prozessorregister

- Ein x86-Prozessor besitzt mehrere Register (interne Speicherbereiche), in denen er benötigte Zahlen und Adressen speichert:
  - **`EIP:`** *Instruction pointer*, zeigt auf die aktuelle Stelle im Programmcode
  - **`EFLAGS:`** *Status-Register*, speichert verschiedene Einstellungen und Informationen
  - **`EBP:`** *Base Pointer*, wird für den Zugriff auf lokale Variablen und Funktionsparameter benötigt
  - **`ESI, EDI:`** *Source- und Destination-Index*, werden bspw. beim Kopieren von Daten verwendet, meist aber zur allgemeinen Verwendung
  - **`EAX, EBX, ECX, EDX:`** Register zur allgemeinen Verwendung

Prozessorregister

- alle diese Register sind 32 Bit breit $\Rightarrow$ Zahlen bis zur Größe von $2^{32} \approx 4,3$ Mrd. (ohne Vorzeichen)
- Die Register für die allgemeine Verwendung können auch als Teilbereiche angesprochen werden (Bsp. **EAX**):



- Delphi `Integer` $\rightarrow$ z. B. **EAX**
- Delphi `Char` $\rightarrow$ z. B. **AL** oder **AH**

Darstellung von negativen Zahlen

- Negative Zahlen werden im Zweierkomplement gespeichert:

→ Bits werden umgekehrt: 0 wird 1, 1 wird 0

→ 1 wird addiert

- 00 00 01 01b (5) wird zu 11 11 10 11b (-5 $\widehat{=}$ 251)
- Zahlenbereich ohne Voreichen kleiner, da das letzte Bit das Vorzeichen ist (nur 2^{31})

→ Zwei 32-Bit-Zahlentypen in Delphi: `Integer` mit Vorzeichen, `Cardinal` ohne Vorzeichen

Speicher

- Aus Sicht des Prozessors gibt es für ein Programm mehrere Speicherbereiche:
  - **Code**-Bereich (Text): Programmcode
  - **Stack**: lokale Variablen, gespeicherte Rücksprungadressen bei Funktionsaufrufen
  - **Datenbereich(e)**: globale Variablen, dynamisch angeforderter Speicher(z. B. für `Strings`)
- Die verschiedenen Bereiche müssen nicht an verschiedenen Stellen liegen

Allgemeines

- Befehle des **x86** sind Bit-Folgen mit unterschiedlicher Länge
- einfache Befehle bestehen aus 1 Byte, andere aus 2 oder 3 Byte
- Zu einigen Befehlen können Konstanten hinzukommen
- Für Menschen unlesbar: `B878563412`
- Einführung von **Mnemonics** = Textkürzel für die einzelnen Befehle
- aus `B878563412` wird **`MOV EAX,`** `12345678`

Registerbefehle

Wert „bewegen“: MOV

MOV EAX, $12345678
→ Lade die (hexadezimale) Zahl 12345678h nach **EAX**

MOV EAX, EDX
→ Lade den Wert von **EDX** nach **EAX**

MOV [EAX], $12345678
→ Schreibe die Zahl 12345678h in den Speicher an die Adresse die in **EAX** steht

Registerbefehle

Werte auf Stack legen: PUSH

PUSH $12345678

→ Pushe 12345678h auf den Stack

PUSH EAX

→ Lege den Inhalt von **EAX** oben auf den Stack

Werte von Stack holen : POP

POP EAX

→ Obersten Wert von Stack nach **EAX** schreiben und vom Stack löschen

Rechenbefehle

Addition: ADD

ADD EAX, 10

→ Addiere 10 zum Wert von **EAX**

Inkrementieren: INC

INC EAX

→ Addiere 1 zum Wert von **EAX**

Rechenbefehle

Subtraktion: SUB

```
SUB EAX, 10
```

→ Subtrahiere 10 vom Wert von **EAX**

Dekrementieren: DEC

```
DEC EAX
```

→ Subtrahiere 1 vom Wert von **EAX**

Rechenbefehle

Multiplikation mit Vorzeichen: `IMUL`

`IMUL EAX, ECX`

→ Multipliziere **EAX** mit **ECX** und schreibe das Ergebnis nach **EAX**

Multiplikation ohne Vorzeichen: `MUL`

`MUL ECX`

→ Multipliziere **EAX** mit **ECX**

(Funktioniert nur in Verbindung mit **EAX**)

Rechenbefehle

Division mit Vorzeichen: `IDIV`

`IDIV ECX`

→ Dividiere **`EDX:EAX`** durch **`ECX`** und schreibe den Quotienten nach **`EAX`** und den Rest nach **`EDX`**

(Der Dividend steht immer in EAX, das Vorzeichen muss in EDX stehen)

Division ohne Vorzeichen: `DIV`

`DIV ECX`

→ Dividiere **`EDX:EAX`** durch **`ECX`** und schreibe den Quotienten nach **`EAX`** und den Rest nach **`EDX`**

(Der Dividend steht immer in EAX, das Vorzeichen muss in EDX stehen)

Logische Befehle

Bitweise UND-Verknüpfung: AND

`AND AL, DL`

→ „Verundet“ **`AL`** mit **`DL`**

⇒ Nur wenn ein Bit in beiden Operanden gesetzt ist, ist es auch im Ergebnis gesetzt

Bitweise ODER-Verknüpfung: OR

`OR AL, DL`

→ „Verodert“ **`AL`** mit **`DL`**

⇒ Wenn ein Bit in einem oder beiden der Operanden gesetzt ist, ist es auch im Ergebnis gesetzt

Logische Befehle

Bitweise Exklusive-ODER-Verknüpfung: XOR

`XOR AL, DL`

$\rightarrow$ Verknüpft **`AL`** und **`DL`** mit exklusivem Oder

$\Rightarrow$ Wenn ein Bit nur in einem der beiden Operanden gesetzt ist, ist es auch im Ergebnis gesetzt

Vergleichsbefehle

Werte nummerisch vergleichen: CMP

Werte werden intern voneinander subtrahiert

```
CMP EAX, $12345678
```

→ Vergleiche 12345678h mit **EAX**

Werte logisch vergleichen: TEST

Werte werden intern „ge***und***et“

```
TEST EAX, 1
```

→ Teste ob in **EAX** Bit 0 gesetzt ist

Sprungbefehle

Unbedingter Sprung: JMP

JMP [EAX]

→ Springe an die Adresse, die in **EAX** steht

JMP `weiter`

→ Springe an die Adresse, die mit `weiter` angegeben wird

Sprungbefehle

Bedingter Sprung: Jxx

xx steht für einen **Condition Code**:

- E: equal ⇒ gleich
- NE: not equal ⇒ ungleich
- G: greather ⇒ größer als
- LE: less or equal ⇒ kleiner oder gleich
- NZ: not zero ⇒ TEST erfolgreich

Beispiel

```
CMP EAX, 1
JG groesser
JL kleiner

TEST EDX, 1
JNZ weiter
```

Funktionsbefehle

Funktionsaufruf: CALL

`CALL` `$12345678`

→Speichere aktuelle Position im Code (**`EIP`**) auf dem Stack und springe nach 12345678h

Funktionsrückkehr: RET

`RET`

→Hole die gesicherte Adresse vom Stack und springe dorthin

Allgemeines

- Umsetzung der Delphi-Konstrukte nach festen Schemata
- Optimierung und Vereinfachung der Befehle im Kontext
- Delphi optimiert nicht besonders gut, ein Mensch könnte den Code teilweise besser optimieren
- Für hohe Optimierung genaue Informationen über Prozessor notwendig

Funktionen – Teil 1

- Erste 3 Parameter in den Registern **EAX**, **EDX**, **ECX**
- Zahlen und Chars stehen direkt in den Registern
- bei `var`-Parametern steht die Adresse der Variable im entsprechenden Register
- Aufruf über einen `CALL`-Befehl
- Register **EBX**, **ESI**, **EDI** und **EBP** müssen am Ende gleichen Wert haben wie am Anfang
- Rückgabewert (`Result`) in **EAX**
- Beenden der Funktion mit `RET`

if-Anweisung

Delphi

```
function if_bsp1
(a, b: Integer)
:Integer;
begin
  if a = b then
  result := 0
  else
  if (a > b) then
    result := 1
  else
    result := -1;
end;
```

if-Anweisung

Delphi

```
function if_bsp1
(a, b: Integer)
:Integer;
begin
  if a = b then
  result := 0
  else
  if (a > b) then
    result := 1
  else
    result := -1;
end;
```

Dissassemblat

```
ifbsp.pas.13: if a = b then
00408968 3BD0              cmp edx,eax
0040896A 7503              jnz $0040896f

ifbsp.pas.14: result := 0
0040896C 33C0              xor eax,eax
            // entspricht mov eax, 0
0040896E C3                ret

ifbsp.pas.16: if (a > b) then
0040896F 3BD0              cmp edx,eax
00408971 7D07              jnl $0040897a

ifbsp.pas.17: result := 1
00408973 B801000000
mov eax,$00000001
00408978 EB03              jmp $0040897d

ifbsp.pas.19: result := -1;
0040897A 83C8FF            or eax,-$01
            // entspricht mov eax, -1
ifbsp.pas.20: end;
0040897D C3                ret
```

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1: cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2: mov eax,-1
@3: ret
```

1. Vergleiche b mit a
2. Wenn a≠b: springe zum nächten Vergleich (@1)
3. Sonst: setzte result auf 0 und beende Funktion
4. Vergleiche b mit a
5. Wenn b ≮ a: springe weiter (@2)
6. Sonst: setzte result auf 1 und springe zum Ende
7. a < b: setzte result auf -1
8. Beenden der Funktion

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1: cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2: mov eax,-1
@3: ret
```

1. Vergleiche `b` mit `a`
2. Wenn `a`≠`b`: springe zum nächten Vergleich (`@1`)
3. Sonst: setzte `result` auf 0 und beende Funktion
4. Vergleiche `b` mit `a`
5. Wenn `b` $\not<$ `a`: springe weiter (`@2`)
6. Sonst: setzte `result` auf 1 und springe zum Ende
7. `a` < `b`: setzte `result` auf -1
8. Beenden der Funktion

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1: cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2: mov eax,-1
@3: ret
```

1. Vergleiche `b` mit `a`
2. Wenn `a`≠`b`: springe zum nächten Vergleich (`@1`)
3. Sonst: setzte `result` auf 0 und beende Funktion
4. Vergleiche `b` mit `a`
5. Wenn `b` $\not<$ `a`: springe weiter (`@2`)
6. Sonst: setzte `result` auf 1 und springe zum Ende
7. `a` < `b`: setzte `result` auf -1
8. Beenden der Funktion

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1: cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2: mov eax,-1
@3: ret
```

1. Vergleiche `b` mit `a`
2. Wenn `a`≠`b`: springe zum nächten Vergleich (`@1`)
3. Sonst: setzte `result` auf 0 und beende Funktion
4. Vergleiche `b` mit `a`
5. Wenn `b` ≮ `a`: springe weiter (`@2`)
6. Sonst: setzte `result` auf 1 und springe zum Ende
7. `a` < `b`: setzte `result` auf -1
8. Beenden der Funktion

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1: cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2: mov eax,-1
@3: ret
```

1. Vergleiche `b` mit `a`
2. Wenn `a`$\neq$`b`: springe zum nächten Vergleich (`@1`)
3. Sonst: setzte `result` auf 0 und beende Funktion
4. Vergleiche `b` mit `a`
5. Wenn `b` $\not<$ `a`: springe weiter (`@2`)
6. Sonst: setzte `result` auf 1 und springe zum Ende
7. `a` < `b`: setzte `result` auf -1
8. Beenden der Funktion

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1: cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2: mov eax,-1
@3: ret
```

1. Vergleiche `b` mit `a`
2. Wenn `a`≠`b`: springe zum nächten Vergleich (`@1`)
3. Sonst: setzte `result` auf 0 und beende Funktion
4. Vergleiche `b` mit `a`
5. Wenn `b` $\not<$ `a`: springe weiter (`@2`)
6. Sonst: setzte `result` auf 1 und springe zum Ende
7. `a` < `b`: setzte `result` auf -1
8. Beenden der Funktion

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1: cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2: mov eax,-1
@3: ret
```

1. Vergleiche `b` mit `a`
2. Wenn `a`$\neq$`b`: springe zum nächten Vergleich (`@1`)
3. Sonst: setzte `result` auf 0 und beende Funktion
4. Vergleiche `b` mit `a`
5. Wenn `b` $\not<$ `a`: springe weiter (`@2`)
6. Sonst: setzte `result` auf 1 und springe zum Ende
7. `a` < `b`: setzte `result` auf -1
8. Beenden der Funktion

if-Anweisung
Schritt für Schritt . . .

Maschinencode

```
    cmp edx,eax
    jne @1
    xor eax,eax
    ret
@1:cmp edx,eax
    jnl @2
    mov eax,1
    jmp @3
@2:mov eax,-1
@3:ret
```

1. Vergleiche `b` mit `a`
2. Wenn `a`$\neq$`b`: springe zum nächten Vergleich (`@1`)
3. Sonst: setzte `result` auf 0 und beende Funktion
4. Vergleiche `b` mit `a`
5. Wenn `b` $\not<$ `a`: springe weiter (`@2`)
6. Sonst: setzte `result` auf 1 und springe zum Ende
7. `a` < `b`: setzte `result` auf -1
8. Beenden der Funktion

`case`-Anweisungen

Zwei sehr verschiedene Möglichkeiten für Umsetzung einer `case`-Anweisung:

- Vergleiche → wie viele `if`s hintereinander
- Sprungtabelle → Tabelle ordnet jeder Zahl ein Sprungziel zu

case mit Vergleichen

Delphi

```
function case_bsp1
(i: Cardinal): Char;
begin
  case i of
  0..5: result:= '-';
  6..10: result:= '0';
  else: result:= '+';
  end;
end;
```

Dissassemblat

```
casebsp.pas.26: case i of
00408A54 83E806           sub eax,$06
00408A57 7207             jb $00408a60
00408A59 83E805           sub eax,$05
00408A5C 7205             jb $00408a63
00408A5E EB06             jmp $00408a66
casebsp.pas.27: 0..5: result:= '-';
00408A60 B02D             mov al,$2d
00408A62 C3               ret
casebsp.pas.28: 6..10: result:= '0';
00408A63 B030             mov al,$30
00408A65 C3               ret
casebsp.pas.29: else result:= '+';
00408A66 B02B             mov al,$2b
casebsp.pas.31: end;
00408A68 C3               ret
```

case mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1: mov al, $2d
    ret
@2: mov al, $30
    ret
@3: mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`< 0: springe nach (@1)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` < 0: springe nach (@2)
5. Sonst: springe zum `else`-Teil (@3)
6. `i` war in $\{0 .. 5\}$: `result` $:=$ '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` $:=$ '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` $:=$ '+' und Ende

`case` mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1: mov al, $2d
    ret
@2: mov al, $30
    ret
@3: mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`<0: springe nach (`@1`)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` < 0: springe nach (`@2`)
5. Sonst: springe zum `else`-Teil (`@3`)
6. `i` war in $\{0 .. 5\}$: `result` $:=$ '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` $:=$ '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` $:=$ '+' und Ende

case mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1: mov al, $2d
    ret
@2: mov al, $30
    ret
@3: mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`<0: springe nach (`@1`)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` < 0: springe nach (@2)
5. Sonst: springe zum `else`-Teil (@3)
6. `i` war in $\{0 .. 5\}$: `result` := '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` := '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` := '+' und Ende

case mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1: mov al, $2d
    ret
@2: mov al, $30
    ret
@3: mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`<0: springe nach (`@1`)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` <0: springe nach (`@2`)
5. Sonst: springe zum `else`-Teil (`@3`)
6. `i` war in $\{0 .. 5\}$: `result` := '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` := '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` := '+' und Ende

case mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1: mov al, $2d
    ret
@2: mov al, $30
    ret
@3: mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`<0: springe nach (`@1`)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` <0: springe nach (`@2`)
5. Sonst: springe zum `else`-Teil (`@3`)
6. `i` war in $\{0 .. 5\}$: `result` := '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` := '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` := '+' und Ende

case mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1: mov al, $2d
    ret
@2: mov al, $30
    ret
@3: mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`<0: springe nach (`@1`)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` <0: springe nach (`@2`)
5. Sonst: springe zum `else`-Teil (`@3`)
6. `i` war in $\{0 .. 5\}$: `result` := '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` := '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` := '+' und Ende

case mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1:mov al, $2d
    ret
@2:mov al, $30
    ret
@3:mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`<0: springe nach (`@1`)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` <0: springe nach (`@2`)
5. Sonst: springe zum `else`-Teil (`@3`)
6. `i` war in $\{0 .. 5\}$: `result` := '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` := '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` := '+' und Ende

case mit Vergleichen
Schritt für Schritt . . .

Maschinencode

```
    sub eax,6
    jb @1
    sub eax,5
    jb @2
    jmp @3
@1:mov al, $2d
    ret
@2:mov al, $30
    ret
@3:mov al, $2b
    ret
```

1. 6 von `i` subtrahieren, da (0..5)
2. Wenn jetzt `i`<0: springe nach (`@1`)
3. 5 von `i` subtrahieren, da (6..10)
4. Wenn jetzt `i` <0: springe nach (`@2`)
5. Sonst: springe zum `else`-Teil (`@3`)
6. `i` war in $\{0 .. 5\}$: `result` := '-' und Ende
7. `i` war in $\{6 .. 10\}$: `result` := '0' und Ende
8. `i` war nicht in $\{0 .. 10\}$: `result` := '+' und Ende

case mit Sprungtabelle

Delphi

```
function case_bsp2
(i: Cardinal): Integer;
begin
  case i of
  1: result:= 6;
  2: result:= 5;
  3: result:= 4;
  4: result:= 3;
  5: result:= 2;
  else result := 1;
  end;
end;
```

Dissassemblat

```
casebsp.pas.13: case i of
00408A08 83F805          cmp eax,$05
00408A0B 773D            jnbe $00408a4a
00408A0D FF2485148A4000
jmp dword ptr [eax*4+$408a14]
00408A14 //[Adresstabelle]
casebsp.pas.14: 1: result:= 6;
00408A2C B806000000      mov eax,6
00408A31 C3              ret
casebsp.pas.15: 2: result:= 5;
00408A32 B805000000      mov eax,5
00408A37 C3              ret
casebsp.pas.16: 3: result:= 4;
00408A38 B804000000      mov eax,4
00408A3D C3              ret
casebsp.pas.17: 4: result:= 3;
00408A3E B803000000      mov eax,3
00408A43 C3              ret
casebsp.pas.18: 5: result:= 2;
00408A44 B802000000      mov eax,2
00408A49 C3              ret
casebsp.pas.19: else result := 1;
00408A4A B801000000      mov eax,1
casebsp.pas.21: end;
00408A4F C3              ret
```

Sprungtabelle

Speicherauszug

```
00408A14 4A 8A 40 00
00408A18 2C 8A 40 00
00408A1C 32 8A 40 00
00408A20 38 8A 40 00
00408A24 3E 8A 40 00
00408A28 44 8A 40 00
```

Sprungtabelle

i	Codestelle	Label
0	00048A4A	@else
1	00048A2C	@1
2	00048A32	@2
3	00048A38	@3
4	00048A3E	@4
5	00048A44	@5

case-Anweisung mit Sprungtabelle

Schritt für Schritt . . .

Maschinencode

```
        cmp eax,$05
        ja @else
        jmp [tabelle+eax*4]
@1:     mov eax,6
        ret
@2:     mov eax, 5
        ret
    ...
@else:  mov eax,1
        ret
```

- Vergleich `i` > 5
- Wenn ja, springe zum `else`-Teil (`@else`)
- Springe an die Adresse, die in der Tabelle für `i` steht
- Weise `result` den entsprechnden Wert zu und beende die Funktion

case-Anweisung mit Sprungtabelle

Schritt für Schritt . . .

Maschinencode

```
        cmp eax,$05
        ja @else
        jmp [tabelle+eax*4]
@1:     mov eax,6
        ret
@2:     mov eax, 5
        ret
    ...
@else:  mov eax,1
        ret
```

- Vergleich `i` > 5
- Wenn ja, springe zum `else`-Teil (`@else`)
- Springe an die Adresse, die in der Tabelle für `i` steht
- Weise `result` den entsprechnden Wert zu und beende die Funktion

case-Anweisung mit Sprungtabelle

Schritt für Schritt . . .

Maschinencode

```
        cmp eax,$05
        ja @else
        jmp [tabelle+eax*4]
@1:     mov eax,6
        ret
@2:     mov eax, 5
        ret
    ...
@else:  mov eax,1
        ret
```

- Vergleich `i` > 5
- Wenn ja, springe zum `else`-Teil (`@else`)
- Springe an die Adresse, die in der Tabelle für `i` steht
- Weise `result` den entsprechnden Wert zu und beende die Funktion

`case`-Anweisung mit Sprungtabelle

Schritt für Schritt . . .

Maschinencode

```
      cmp eax,$05
      ja @else
      jmp [tabelle+eax*4]
@1:   mov eax,6
      ret
@2:   mov eax, 5
      ret
   ...
@else:mov eax,1
      ret
```

- Vergleich `i` > 5
- Wenn ja, springe zum `else`-Teil (`@else`)
- Springe an die Adresse, die in der Tabelle für `i` steht
- Weise `result` den entsprechnden Wert zu und beende die Funktion

Schleifen

Die verschiedenen Schleifen in Delphi sind von ihrem Aufbau sehr ähnlich:

- Überprüfen einer Bedingung
- Schleifencode ausführen oder nach Schleife weitermachen
- Innerhalb der Schleife: Veränderung der Parameter für die Bedingung: sonst **Endlosschleife**!
- Je nach Schleifentyp: Überprüfung der Bedingung zu verschiedenen Zeiten (vor oder nach Ausführung des Schleifencodes)

while-Schleife

Delphi

```
function while_bsp1
(a, b: Integer):Integer;
begin
  result := 0;
  while a < b do
    begin
      a := a * a;
      inc(result);
    end;
end;
```

Dissassemblat

```
schleifen.pas.30: result := 0;
00408A7C 33C9             xor ecx,ecx
schleifen.pas.31: while a < b do
00408A7E 3BD0             cmp edx,eax
00408A80 7E08             jle $00408a8a
schleifen.pas.33: a := a * a;
00408A82 0FAFC0           imul eax,eax
schleifen.pas.34: inc(result);
00408A85 41               inc ecx
schleifen.pas.31: while a < b do
00408A86 3BD0             cmp edx,eax
00408A88 7FF8             jnle $00408a82
schleifen.pas.36: end;
00408A8A 8BC1             mov eax,ecx
00408A8C C3               ret
```

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
    xor ecx, ecx
    cmp edx,eax
    jle @e
@s: imul eax,eax
    inc ecx
    cmp edx,eax
    jg @s
@e: mov eax,ecx
    ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **`EAX`** und Ende des Funktion

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
    xor ecx, ecx
    cmp edx,eax
    jle @e
@s: imul eax,eax
    inc ecx
    cmp edx,eax
    jg @s
@e: mov eax,ecx
    ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **`EAX`** und Ende des Funktion

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
    xor ecx, ecx
    cmp edx,eax
    jle @e
@s: imul eax,eax
    inc ecx
    cmp edx,eax
    jg @s
@e: mov eax,ecx
    ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
    xor ecx, ecx
    cmp edx,eax
    jle @e
@s:imul eax,eax
    inc ecx
    cmp edx,eax
    jg @s
@e:mov eax,ecx
    ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
    xor ecx, ecx
    cmp edx,eax
    jle @e
@s:imul eax,eax
    inc ecx
    cmp edx,eax
    jg @s
@e:mov eax,ecx
    ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
    xor ecx, ecx
    cmp edx,eax
    jle @e
@s:imul eax,eax
    inc ecx
    cmp edx,eax
    jg @s
@e:mov eax,ecx
    ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
    xor ecx, ecx
    cmp edx,eax
    jle @e
@s: imul eax,eax
    inc ecx
    cmp edx,eax
    jg @s
@e: mov eax,ecx
    ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

while-Schleife
Schritt für Schritt . . .

Maschinencode

```
   xor ecx, ecx
   cmp edx,eax
   jle @e
@s:imul eax,eax
   inc ecx
   cmp edx,eax
   jg @s
@e:mov eax,ecx
   ret
```

1. `result` auf 0 setzen
2. Vergleich ob `b` $\leq$ `a`
3. Wenn ja überspringe Schleife
4. Quadriere `a`
5. Erhöhe `result`
6. Vergleich ob `b` $>$ `a`
7. Wenn ja springe zum Start des Schleifencodes (`@s`)
8. `result` nach **`EAX`** und Ende des Funktion

for-Schleife

Delphi

```
function for_bsp3
(b, n: Integer): Integer;
var i : Cardinal;
begin
  result := b;
  for i := 2 to n do
    result := result * b;
end;
```

Dissassemblat

```
schleifen.pas.32: result := b;
00408A68 8BC8             mov ecx,eax
schleifen.pas.33: for i := 2 to n do
00408A6A 83EA02           sub edx,$02
00408A6D 7207             jb $00408a76
00408A6F 42               inc edx
schleifen.pas.34: result := result * b;
00408A70 0FAFC8           imul ecx,eax
schleifen.pas.33: for i := 2 to n do
00408A73 4A               dec edx
00408A74 75FA             jnz $00408a70
schleifen.pas.35: end;
00408A76 8BC1             mov eax,ecx
00408A78 C3               ret
```

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s: imul ecx,eax
    dec edx
    jnz @s
@e: mov eax,ecx
    ret
```

1. `result` auf `b` setzen
2. `i` auf `n`−2 setzen, da rückwärts gezählt wird
3. Wenn `i` < 0: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i`≠ 0: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **`EAX`** und Ende des Funktion

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s: imul ecx,eax
    dec edx
    jnz @s
@e: mov eax,ecx
    ret
```

1. `result` auf `b` setzen
2. `i` auf `n`-2 setzen, da rückwärts gezählt wird
3. Wenn `i` $<$ 0: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i`$\neq$ 0: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **`EAX`** und Ende des Funktion

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s: imul ecx,eax
    dec edx
    jnz @s
@e: mov eax,ecx
    ret
```

1. `result` auf `b` setzen
2. `i` auf `n`-2 setzen, da rückwärts gezählt wird
3. Wenn `i` $<$ `0`: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i`$\neq$ `0`: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **`EAX`** und Ende des Funktion

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s: imul ecx,eax
    dec edx
    jnz @s
@e: mov eax,ecx
    ret
```

1. `result` auf `b` setzen
2. `i` auf $n-2$ setzen, da rückwärts gezählt wird
3. Wenn `i` $<$ 0: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i` $\neq$ 0: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s:imul ecx,eax
    dec edx
    jnz @s
@e:mov eax,ecx
    ret
```

1. `result` auf `b` setzen
2. `i` auf $n-2$ setzen, da rückwärts gezählt wird
3. Wenn `i` $<$ 0: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i` $\neq$ 0: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s:imul ecx,eax
    dec edx
    jnz @s
@e:mov eax,ecx
    ret
```

1. `result` auf `b` setzen
2. `i` auf `n`−2 setzen, da rückwärts gezählt wird
3. Wenn `i` < 0: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i`≠ 0: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s: imul ecx,eax
    dec edx
    jnz @s
@e: mov eax,ecx
    ret
```

1. `result` auf `b` setzen
2. `i` auf `n`−2 setzen, da rückwärts gezählt wird
3. Wenn `i` < 0: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i`≠ 0: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **EAX** und Ende des Funktion

for-Schleife
Schritt für Schritt . . .

Maschinencode

```
    mov ecx,eax
    sub edx,$02
    jb @e
    inc edx
@s:imul ecx,eax
    dec edx
    jnz @s
@e:mov eax,ecx
    ret
```

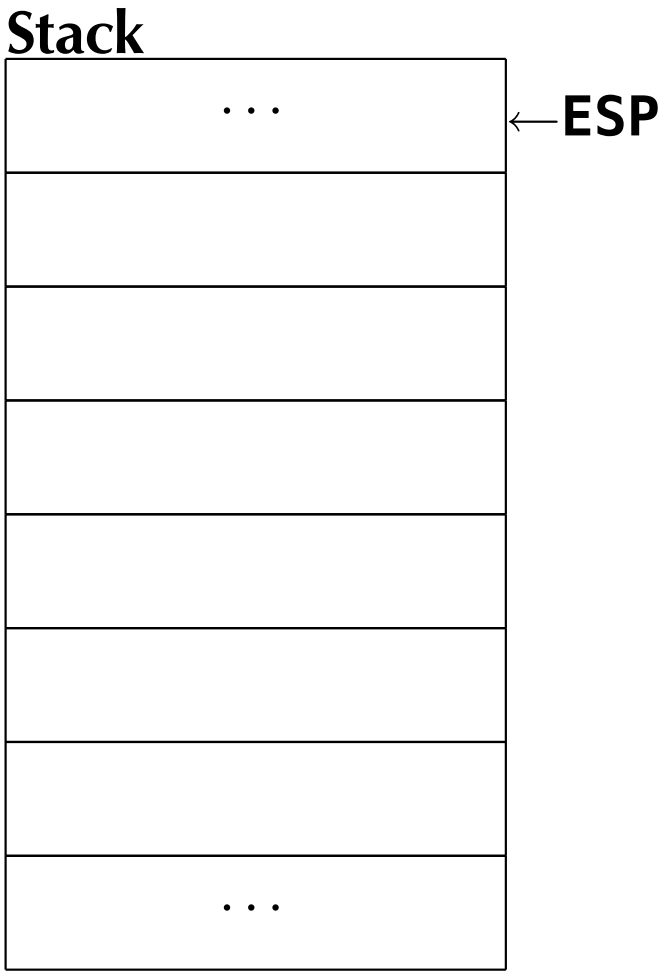
1. `result` auf `b` setzen
2. `i` auf $n-2$ setzen, da rückwärts gezählt wird
3. Wenn `i` $<$ 0: überspringe Schleife
4. Erhöhe `i` um 1
5. Multipliziere `result` mit `b`
6. Erniedrige `i` um 1
7. Wenn `i`$\neq$ 0: springe zum Start des Schleifencodes (`@s`)
8. `result` nach **`EAX`** und Ende des Funktion

Funktionen – Teil 2

- Alle Parameter ab dem vierten landen auf dem **Stack**
- für jeden zusätzlichen Parameter ein **PUSH** `variable`
- Zugriff wird über Register **EBX** realisiert
- **Lokale Variablen** ebenfalls auf dem Stack und Zugriff über **EBX**

Stackverwendung von Funktionen

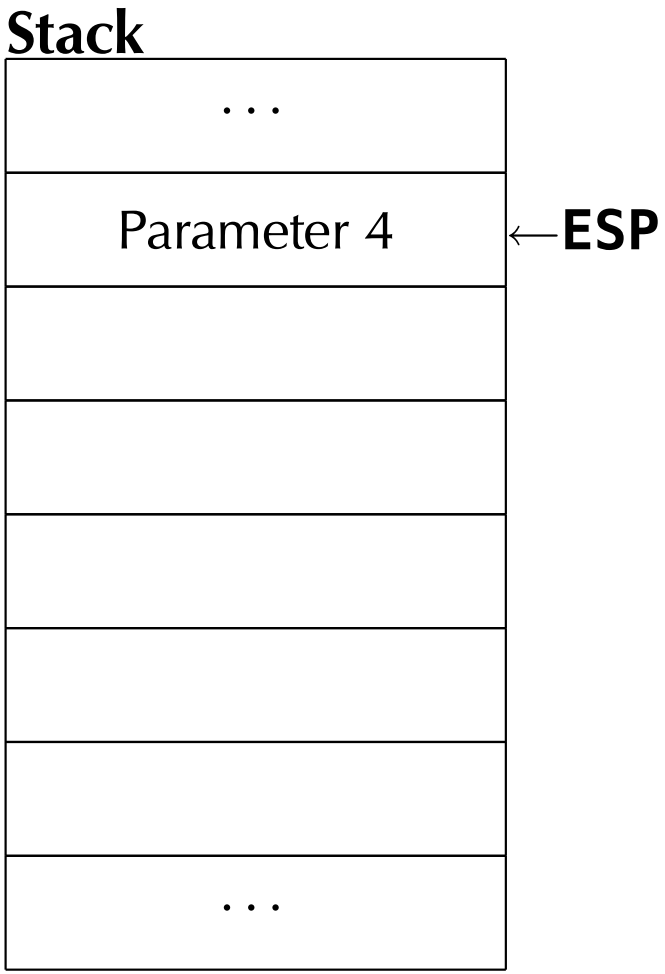
am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen



Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

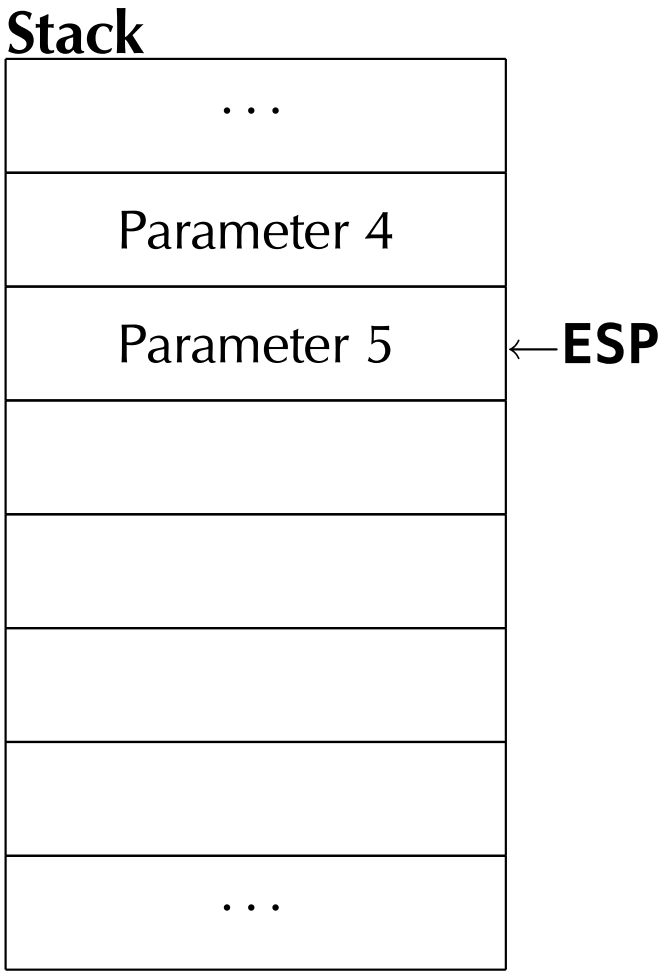
Stackverwendung von Funktionen
am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen



Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen
am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen

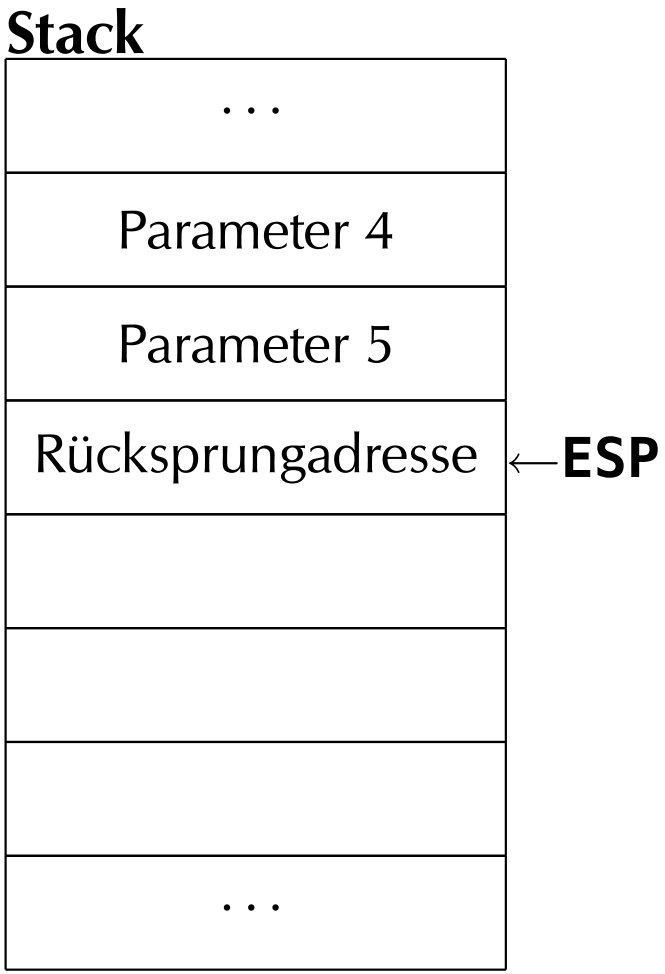


Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen

am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen

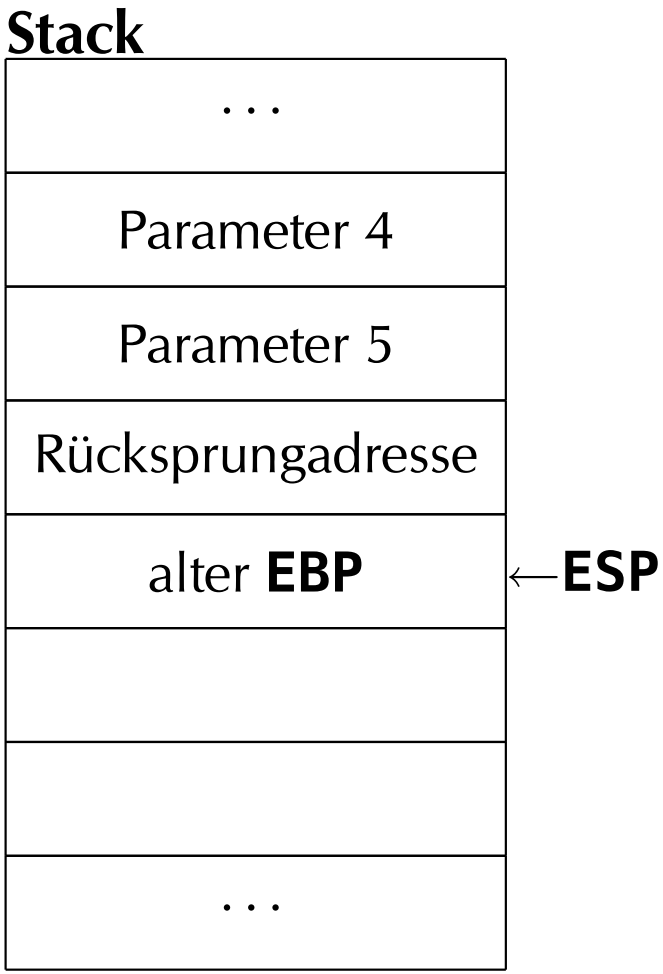


Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen

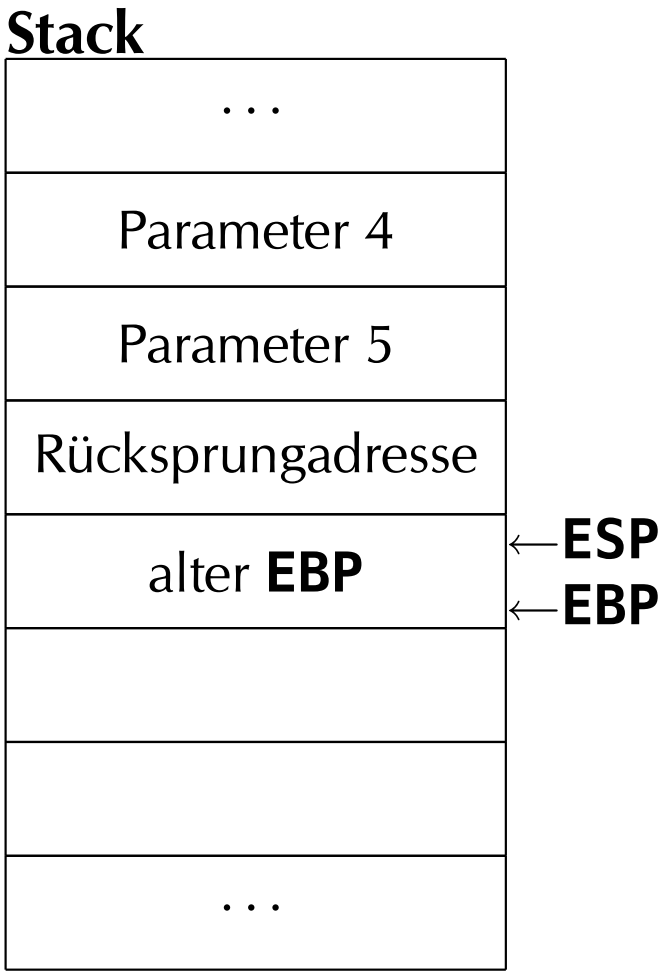
am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen



Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen
am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen

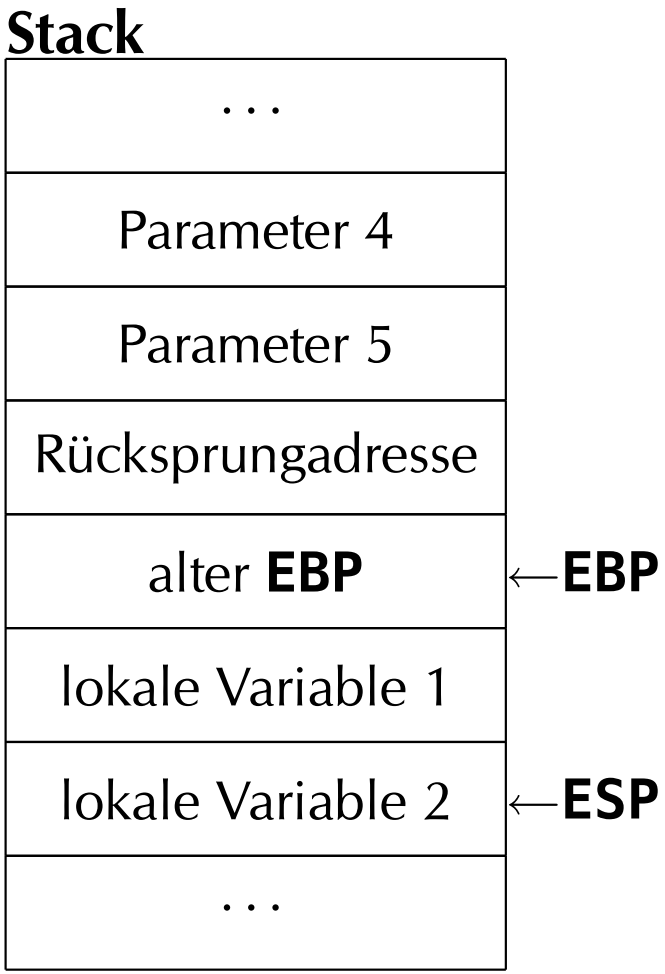


Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen

am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen

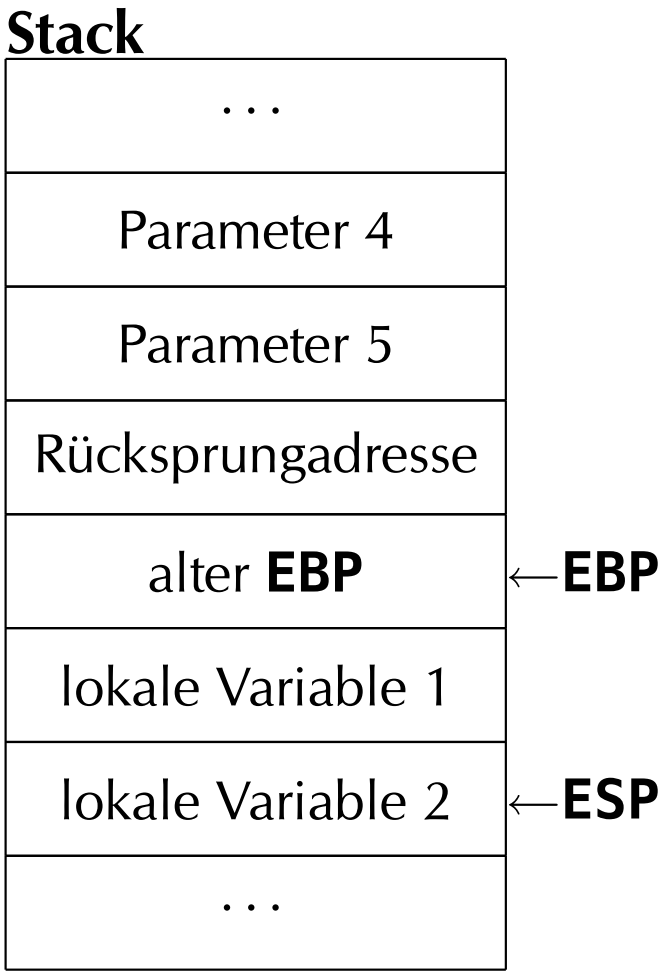


Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen

am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen

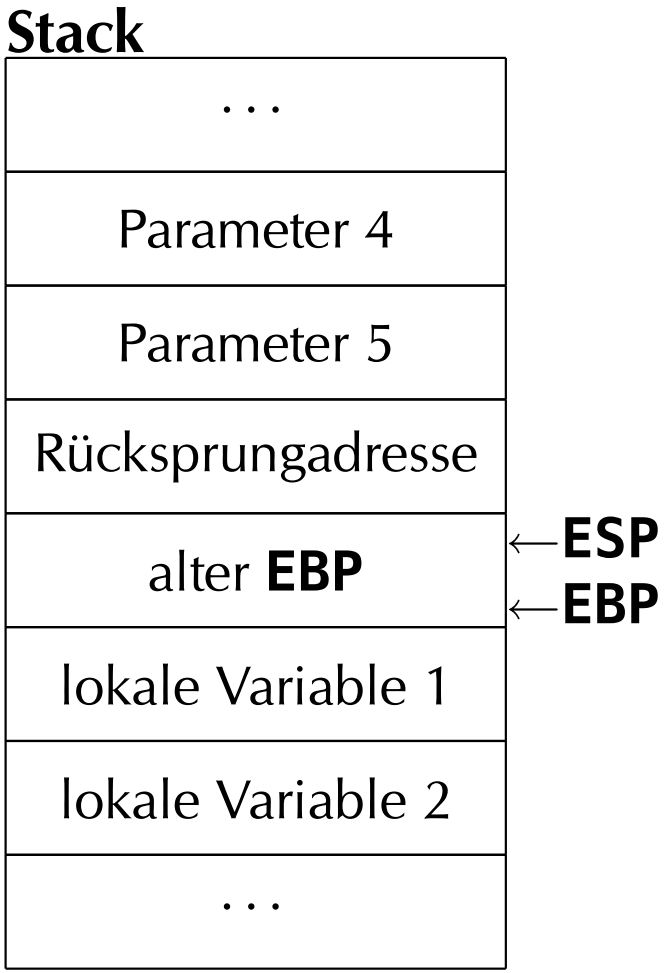


Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen

am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen

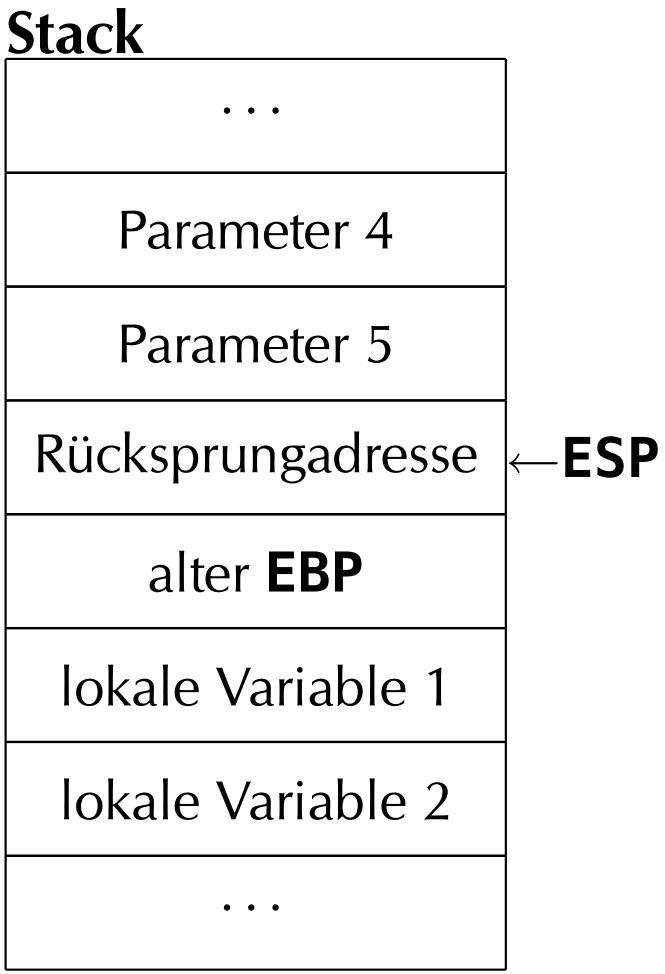


Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen

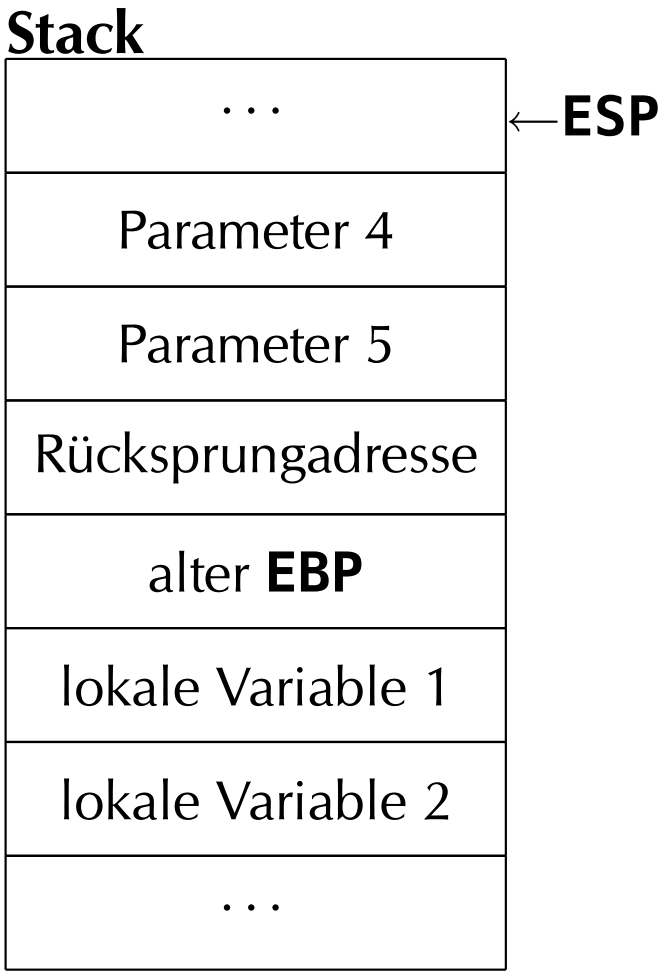
am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen



Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen
am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen

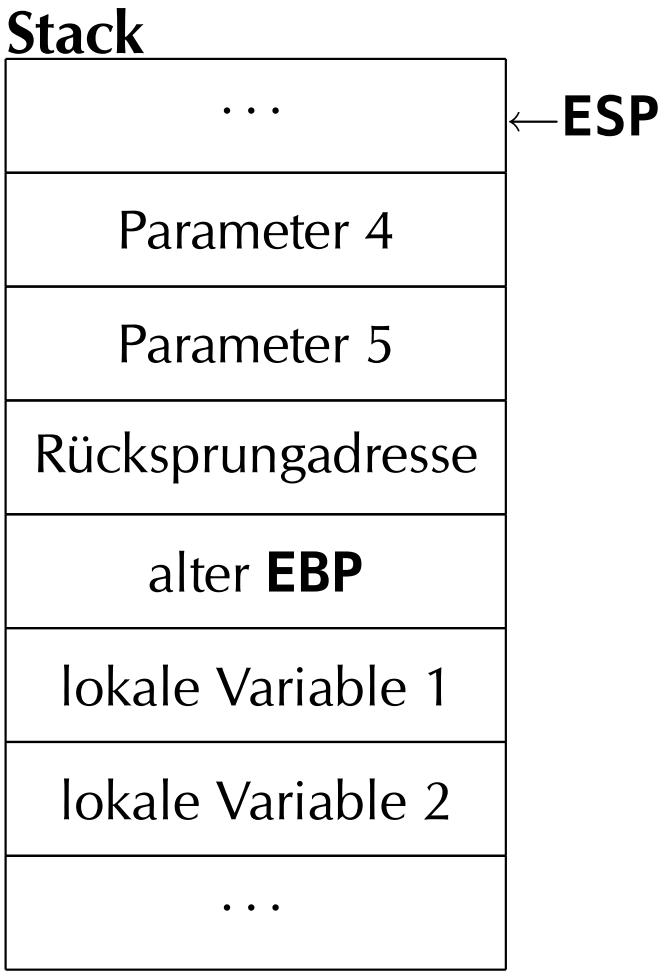


Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Stackverwendung von Funktionen

am Beispiel einer Funktion mit 5 Parametern und 2 lokalen Variablen



Beispielcode

```
...
mov  ecx, param3
push param4
push param5
call funktion
push ebp
mov  ebp, esp
sub esp, 8
mov  [ebp-4], 1  // Variable 1
add  eax, [ebp+8]// Parameter 5
...
mov  esp, ebp
pop  ebp
ret 8
...
```

Quellen

- ASSEMBLER GE-PACKT, Joachim Rohde, mitip-Verlag, 2. Auflage 2007
- 80386-HANDBUCH, Penn Brumm, Markt & Technik Verlag, 1. Auflage 1989
- Hilfetexte von BORLAND TURBO DELPHI 2006
- http://de.wikipedia.org/wiki/8086
- http://de.wikipedia.org/wiki/80386
- http://de.wikipedia.org/wiki/I386
- http://de.wikipedia.org/wiki/X86

Ende