

## x86-Assembler Referenz

Eine kleine Übersicht über die gebräuchlichsten Assembler-Befehle:

Befehl	Bedeutung	Pseudocode	Beispiel
movl von, nach	Schiebt den Wert „von“ nach „nach“	eax = ebx;	movl %ebx, %eax
pushl wert	Schiebt den Wert „wert“ auf den Stack	eax = 4;	movl \$4, %eax
popl register	Schreibt den letzten Wert des Stacks nach „register“	esp--;	pushl %ebx
		* (esp) = wert;	popl %ebx
		register = *esp;	
		esp++;	
cmp b, a	Vergleicht „b“ mit „a“, speichert das Resultat in den Prozessorflags		cmp \$5, %eax
jmp ziel	Springe nach „ziel“	goto ziel;	jmp ende
je ziel	Springe nach „ziel“, wenn a = b	if (a==b) goto ziel;	je gleich
jne ziel	Springe nach „ziel“, wenn a != b	if (a!=b) goto ziel;	jne ungleich
jl ziel	Springe nach „ziel“, wenn a < b (Signiert)	if (a<b) goto ziel;	jl kleiner
jg ziel	Springe nach „ziel“, wenn a > b (Signiert)	if (a>b) goto ziel;	jg groesser
jle ziel	Springe nach „ziel“, wenn a <= b (Signiert)	if (a<=b) goto ziel;	jle kleinergl
jge ziel	Springe nach „ziel“, wenn a >= b (Signiert)	if (a>=b) goto ziel;	jge groessergl
ja ziel	Springe nach „ziel“, wenn a > b (Unsigniert)	if (a>b) goto ziel;	ja groesser
jb ziel	Springe nach „ziel“, wenn a < b (Unsigniert)	if (a<b) goto ziel;	jb kleiner
jae ziel	Springe nach „ziel“, wenn a >= b (Unsigniert)	if (a>=b) goto ziel;	jae groessergl
jbe ziel	Springe nach „ziel“, wenn a <= b (Unsigniert)	if (a<=b) goto ziel;	jbe kleinergl
incl register	Inkrementiert Regsiter „register“	eax++;	incl %eax
decl register	Dekrementiert Regsiter „register“	eax--;	decl %eax
addl was, zu	Addiert den Wert „was“ zu „zu“	eax += ebx;	addl %ebx, %eax
subl was, von	Subtrahiert den Wert „was“ von „von“	eax -= ebx;	subl %ebx, %eax
mull wert	Multipliziert „wert“ mit %eax (Unsigniert) Resultat nach %edx:%eax (Highbyte, Lowbyte)	edx = (ebx*eax)>>32;	mull %ebx
imull wert	Multipliziert „wert“ mit %eax (Signiert) Resultat nach %edx:%eax (Highbyte, Lowbyte)	edx = (ebx*eax)>>32;	imull %ebx
divl wert	Dividiert %eax durch „wert“ (Unsigniert) Resultat nach %eax, Rest nach %edx	eax = (ebx*eax);	divl %ebx
idivl wert	Dividiert %eax durch „wert“ (Signiert) Resultat nach %eax, Rest nach %edx	edx = (eax%ebx);	idivl %ebx
		eax = (eax/ebx);	
andl was, zu	Bitweises UND von „was“ zu „zu“	eax &= ebx;	andl %ebx, %eax
orl was, zu	Bitweises ODER von „was“ zu „zu“	eax  = ebx;	orl %ebx, %eax
xorl was, zu	Bitweises exklusives ODER von „was“ zu „zu“	eax ^= ebx;	xorl %ebx, %eax
notl was	Bitweises NICHT von „was“	eax = ~eax;	notl %eax
shll um, was	Bitweises Shift-Left von „was“ um „um“ Bits	eax <<= 5;	shll \$5, %eax
shrl um, was	Bitweises Shift-Right von „was“ um „um“ Bits	eax >>= 5;	shrl \$5, %eax
sall um, was	Arithmetischer Shift-Left von „was“ um „um“ Bits		sall \$5, %eax
sarl um, was	Arithmetischer Shift-Right von „was“ um „um“ Bits		sarl \$5, %eax
roll um, was	Rotiere links von „was“ um „um“ Bits		roll \$5, %eax
rorl um, was	Rotiere rechts von „was“ um „um“ Bits		rorl \$5, %eax
rcll um, was	Rotiere links von „was“ um „um“ Bits mit Carry		rcll \$5, %eax
rcrl um, was	Rotiere rechts von „was“ um „um“ Bits mit Carry		rcrl \$5, %eax

Themengebiet: „x86-Assembler Referenz (Version 1)“  
Seite: 2  
Copyright: Johannes Bauer (www.johannes-bauer.com)

So sieht normalerweise ein Funktionsaufruf aus:

```
1 Unterprogramm :
2     pushl %ebp          // Basepointer sichern
3     movl %esp, %ebp    // Stackframe erzeugen
4
5     // Jetzt liegen die Parameter an folgenden Adressen:
6     // Parameter1 = 8(%ebp)
7     // Parameter2 = 12(%ebp)
8     // Parameter3 = 16(%ebp)
9
10    // Programmverlauf
11    // ...
12
13    leave              // Stackpointer zurücksetzen (movl %ebp, %esp)
14 ret                 // Zurückkehren
15
16 main:
17     // ...Hauptprogramm...
18
19     pushl Parameter3
20     pushl Parameter2
21     pushl Parameter1
22     call Unterprogramm
23
24     // ...Und weiter...
25 ret
```

Und so eine for-Schleife, die von 7 bis 13 (beides einschließlich!) zählt:

```
1     movl $7, %ecx      // Counter initialisieren
2
3     for_begin:
4         cmpl $13, %ecx // Counter zu groß?
5         jg for_end     // -> Dann for-Schleife beenden
6
7         // Code hier
8
9         incl %ecx      // Counter erhöhen
10    jmp for_begin
11    for_end:
```