Übersicht der wichtigsten Assemblerbefehle in ATT-Syntax

Autoren: Wolfgang Heenes, Patrik Schmittat
Version: 0.4
Datum: 26. März 2011

Operanden, Statusflags und Registersatz

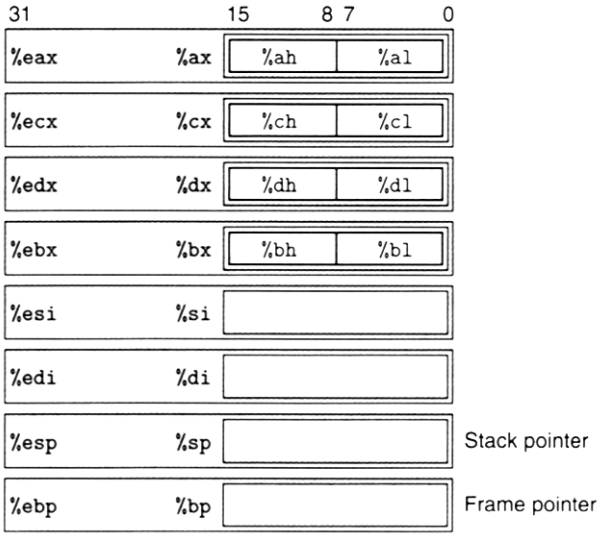
Die vier Operanden sind:

- Direkter Wert (I)
- Register (R)
- Speicher (M)
- Adresse (PTR)

Bei den Befehlen ist angegeben, welche Operanden zulässig sind. Die Befehle sind von der Struktur **Befehl Quelle, Ziel**. Zu jedem Befehl sind Beispiele angegeben.
Außerdem werden für jeden Befehl die Statusflags angegeben. Bei den Flags wird nur noch der erste Buchstabe angegeben. Falls ein Befehl ein Statusbit fest setzt, ist dies via Zuweisung des Flags dargestellt.

- C Übertragsflag (Carryflag)
- Z Nullflag (Zeroflag)
- S Vorzeichenflag (Signflag)
- O Überlaufflag (Overflowflag)

Die anderen Flags spielen im Rahmen dieser Veranstaltung keine Rolle. Die in {..}-Klammern angefügten Symbole deuten auf die möglichen Ausführungen dieses Befehls hin (l = long, w = word, b = byte). Weiterhin ist der Ausdruck Mem[X] als Zugriff auf die Speicheradresse X zu verstehen.



Transportbefehle

Die Adressierung mit zwei M Operanden (z. B. **Befehl M,M**) ist nicht zulässig.

Befehl	Operanden	Statusbits	Beispiel und dessen Wirkung	
`lea{lwb}`	M, R/M	Keine	`leal a, %esi`	%esi := Adresse von a
`mov{lwb}`	I/R/M, R/M	Keine	`movl $0, %eax` `movl %ebp, %esp`	%eax := 0 %esp := %ebp
`pop{lwb}`	R/M	Keine	`popl a`	Legt den Inhalt vom Stack in die Speicheradresse a
`push{lwb}`	R/M	Keine	`pushl a`	Legt den Inhalt der Speicheradresse a auf den Stack
`xchg{lwb}`	R/M, R/M	Keine	`xchgl %eax, %ebx`	Vertauscht die Registerinhalte

Arithmetische Befehle

Die Adressierung mit zwei M Operanden (z. B. **Befehl M,M**) ist nicht zulässig.

Befehl	Operanden	Statusbits	Beispiel und dessen Wirkung	
`adc{lwb}`	I/R/M, R/M	C, Z, S, O	`adcl %ebx, %esp`	%esp := %ebx + C
`add{lwb}`	I/R/M, R/M	C, Z, S, O	`addl $13, %edx`	%edx := %edx + 13
`cdq`		Keine	`cdq`	Verdoppelt die Größe von %eax durch Vorzeichenerweiterung in %edx
`dec{lwb}`	R/M	Z, S, O	`decl %ebx`	%ebx := %ebx - 1
`div{lwb}`	R/M	C, Z, S, O	`divl %ebx`	%eax := %edx:%eax / %ebx (vorzeichenlos, Rest in %edx)
`idiv{lwb}`	R/M	C, Z, S, O	`idivl %ebx`	%eax := %edx:%eax / %ebx (vorzeichenbehaftet, Rest in %edx)
`imul{lwb}`	R/M	C, O	`imull %ecx`	%edx:%eax := %eax * %ecx (vorzeichenbehaftet)
`imul{lwb}`	I/R/M, R	C, O	`imull %ecx, %ebx`	%ebx := %ebx * %ecx (vorzeichenbehaftet)
`inc{lwb}`	R/M	Z, S, O	`incl %eax`	%eax := %eax +1
`mul{lwb}`	R/M	C, O	`mull %ecx`	%edx:%eax := %eax * %ecx (vorzeichenlos)
`neg{lwb}`	R/M	C, Z, S, O	`negl %ebx`	2K-Negation
`sub{lwb}`	I/R/M, R/M	C, Z, S, O	`subl %eax, %edx`	%edx := %edx - %eax

Logische Befehle

Die Adressierung mit zwei M Operanden (z. B. **Befehl M,M**) ist nicht zulässig.
Hinweis: Die Shiftbefehle arbeiten bei der Angabe von *Shiftweiten in Registern* mit dem Register %cl.

Bedeutung der Symbole:

- ! : not
- & : and
- | : or

Befehl	Operanden	Statusbits	Beispiel und dessen Wirkung	
`and{lwb}`	I/R/M, R/M	C=`0`, Z, S, O=`0`	`andl $0xFF, %eax`	%eax := %eax & 0xFF
`not{lwb}`	R/M	Keine	`notl %edx`	%edx := !%eax
`or{lwb}`	I/R/M, R/M	C=`0`, Z, S, O=`0`	`orl %eax, %ebx`	%ebx := %ebx | %eax
`sal{lwb}`	1/I/%cl, R/M	C, Z, S, O	`sall $1, %eax`	Arithmetischer Shift von %eax nach links
`sar{lwb}`	1/I/%cl, R/M	C, Z, S, O	`sarl $1, %eax`	Arithmetischer Shift von %eax nach rechts
`shl{lwb}`	1/I/%cl, R/M	C, Z, S, O	`shll $1, %eax`	Logischer Shift von %eax nach links
`shr{lwb}`	1/I/%cl, R/M	C, Z, S, O	`shrl $1, %eax`	Logischer Shift von %eax nach rechts
`xor{lwb}`	I/R/M, R/M	C=`0`, Z, S, O=`0`	`xorl %eax, %eax`	%eax := %eax ^%eax

Kontroll- und Sprungbefehle

Die Adressierung mit zwei M Operanden (z. B. **Befehl M,M**) ist nicht zulässig. Die Sprungbedingungen sind in der Notation der Intel-Referenz entnommen.

Befehl	Operanden	Statusbits	Beispiel und dessen Wirkung	
`call`	R/M/PTR	C, Z, S, O	`call printf`	Symbol printf aufrufen
`cmp{lwb}`	I/R/M, R/M	C, Z, S, O	`cmpl $0, %eax`	%eax mit 0 vergleichen
`int`	I	Keine	`int $0x80`	Syscall aufrufen
`jmp`	R/M/PTR	C, Z, S, O	`jmp .L1`	Zum Label .L1 springen
`j-`	LBL	Keine		
`je`			`je .L1`	Springt zum Label .L1 falls Z=1
`jne`			`jne .L1`	Springt zum Label .L1 falls Z=`0`
`js`			`js .L1`	Springt zum Label .L1 falls S=1
`jns`			`jns .L1`	Springt zum Label .L1 falls S=`0`
`jg`			`jg .L1`	Springt zum Label .L1 falls Z=`0` & S=O
`jge`			`jge .L1`	Springt zum Label .L1 falls S=O
`jl`			`jl .L1`	Springt zum Label .L1 falls S!=O
`jle`			`jle .L1`	Springt zum Label .L1 falls Z=1 | S!=O
`ja`			`ja .L1`	Springt zum Label .L1 falls C=`0` & Z=`0`
`jae`			`jae .L1`	Springt zum Label .L1 falls C=`0`
`jb`			`jb .L1`	Springt zum Label .L1 falls C=1
`jbe`			`jbe .L1`	Springt zum Label .L1 falls C=1 | Z=1
`ret`	-/I	Keine	`ret`	Verlässt Unterprogramm

Floating-Point Unit

Bei der Ausführung der FPU-Befehle sind die Operanden der arithmetischen Ausdrücke vertauscht und ähneln der **Intel-Notation**. %ST(i) bezeichnet hierbei eines der acht verschiedenen FPU-Register. Die Adressierung zweier %ST(i) Register mit i>0 (z. B. **Befehl %ST(3),%ST(2)**) ist nicht zulässig. Die Operationen fpush und fpop manipulieren die FPU-Register nach der in der Vorlesung vorgestellten Methode.

Befehl	Operanden	Beispiel und dessen Wirkung	
`fadd{l}`	M	`faddl (%esp)`	%ST(0) := %ST(0) + Mem[%esp]
`fadd`	%ST(0)/%ST(i), %ST(0)/%ST(i)	`fadd %ST(0), %ST(3)`	%ST(3) := %ST(0) + %ST(3)
`faddp`	%ST(0), %ST(i)	`faddp %ST(0), %ST(4)`	%ST(4) := %ST(0) + %ST(4); fpop
`faddp`		`faddp`	%ST(1) := %ST(0) + %ST(1); fpop
`fdiv{l}`	M	`fdiv var1`	%ST(0) := %ST(0) / var1
`fdiv`	%ST(0)/%ST(i), %ST(0)/%ST(i)	`fdiv %ST(0), %ST(7)`	%ST(7) := %ST(0) / %ST(7)
`fdivp`	%ST(0), %ST(i)	`fdivp %ST(0), %ST(6)`	%ST(6) := %ST(0) / %ST(6); fpop
`fdivp`		`fdivp`	%ST(1) := %ST(0) / %ST(1); fpop
`fild{l}`	M	`fildl (%esp)`	fpush inttodouble(Mem[%esp])
`fist{l}`	M	`fistl (%esp)`	Mem[%esp] := doubletoint(%ST(0))
`fistp{l}`	M	`fistp var1`	var1 := doubletoint(%ST(0)); fpop
`fld{l}`	M	`fldl (%esp)`	fpush Mem[%esp]
`fld1`		`fld1`	fpush +1.0
`fldz`		`fldz`	fpush +0.0
`fmul{l}`	M	`fmul var1`	%ST(0) := %ST(0) * var1
`fmul`	%ST(0)/%ST(i), %ST(0)/%ST(i)	`fmul %ST(0), %ST(7)`	%ST(7) := %ST(0) * %ST(7)
`fmulp`	%ST(0), %ST(i)	`fmulp %ST(0), %ST(6)`	%ST(6) := %ST(0) * %ST(6); fpop
`fmulp`		`fmulp`	%ST(1) := %ST(0) * %ST(1); fpop
`fst{l}`	M	`fstl (%esp)`	Mem[%esp] := %ST(0)
`fst`	%ST(i)	`fst %ST(2)`	%ST(2) := %ST(0)
`fstp{l}`	M	`fstp var1`	var1 := %ST(0); fpop
`fsub{l}`	M	`fsubl (%esp)`	%ST(0) := %ST(0) - Mem[%esp]
`fsub`	%ST(0)/%ST(i), %ST(0)/%ST(i)	`fsub %ST(0), %ST(3)`	%ST(3) := %ST(0) - %ST(3)
`fsubp`	%ST(0), %ST(i)	`fsubp %ST(0), %ST(4)`	%ST(4) := %ST(0) - %ST(4); fpop
`fsubp`		`fsubp`	%ST(1) := %ST(0) - %ST(1); fpop
`fxch`	%ST(i)	`fxch %ST(6)`	Vertauscht Registerinhalte von %ST(6) und %ST(0)
`fxch`		`fxch`	Vertauscht Registerinhalte von %ST(1) und %ST(0)

Weiterführende Referenzen

Die Referenzen liegen im SVN (SVN-Server, Material ⇒ Referenz ⇒ Intel) bzw. bei `www.intel.com`. Die Dokumente sind:

- Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1
- Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A
- Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B

In diesen Dokumenten wird allerdings Intel-Syntax verwendet.