Der Assembler-Befehlssatz

Anmerkung:
Die kurzen Beispiele beziehen sich auf die Arbeit mit dem DEBUG-Assembler (links) sowie dem MASM, der symbolische Adressen verarbeiten kann (rechts). Es sind nur die Befehle in alphabetischer Reihenfolge aufgeführt, die alle Prozessoren der INTEL-80(X)86-Familie, incl. des 8086/88, beherrschen.

AAA *Adjust Result for ASCII-Addition*

ASCII-Anpassung nach einer Addition

Syntax: `AAA`

Beschreibung:
`AAA` wandelt das AL-Register in eine ungepackte BCD-Zahl um. Sofern die vier unteren Bits von AL kleiner gleich 9 sind, werden die vier oberen Bits auf Null gesetzt und die Flags AF und CF gelöscht. Falls die unteren vier Bits eine ungültige BCD-Zahl (Pseudotetrade) darstellen oder das Halfcarry-Flag (A) gesetzt ist, wird 6 zum AL-Register addiert, AH um eins erhöht, um den Überlauf anzuzeigen, und die vier oberen Bits von AL werden gelöscht. Damit enthält das AX-Register eine 2-stellige BCD Zahl (höherwertiger Teil in AH, niederwertiger Teil in AL).

Beispiel:
Aus `AL=0F`(hex) wird `AX=0105` (=15 BCD)

AAD *Adjust Register for Division*

ASCII-Anpassung für die Division

Syntax: `AAD`

Beschreibung:
`AAD` wandelt eine zweistellige ungepackte BCD-Zahl im AX-Register (höherwertiger Teil in AH, niederwertiger Teil in AL) in die entsprechende Dualzahl um. Dies ist z.B. zur Vorbereitung einer korrekten BCD-Division nötig.

Beispiel:
Aus `AX=0105` (=15 BCD) wird `AX=000F`(hex)

AAM *Adjust Result of BCD-Multiplication*

ASCII-Anpassung für die Multiplikation

Syntax: `AAM`

Beschreibung:
`AAM` wandelt nach einer Multiplikation der Register AL und AH, sofern in beiden Registern eine ungepackte BCD-Zahl stand, das Ergebnis in eine gültige ungepackte BCD-Zahl im AX-Register um.

Beispiel:

```
AX=0909h      (=99 BCD)
MUL AH,AL     (=51 hex)
AAM      ->   AX=0801 (=81 BCD)
```

AAS *Adjust Result for ASCII-Subtraction*

ASCII-Anpassung für die Subtraktion

Syntax: `AAS`

Beschreibung:
`AAS` wandelt die Zahl im AL-Register (z.B. das Ergebnis nach einer Subtraktion) in eine BCD-Zahl um. Sofern die Zahl in AL größer als 9 ist (ungültige ungepackte BCD-Zahl), wird 6 vom AL- und 1 vom AH-Register subtrahiert und die Flags C und A gesetzt, ansonsten werden diese Flags gelöscht. Die höherwertigen 4 Bits im AL-Register werden in jedem Fall gelöscht. Als Ergebnis bleibt eine 4-Bit BCD-Zahl im AL-Register.

ADC *Add with Carry-Flag*

Addieren mit Übertrag

Syntax: `ADC <Zieloperand>,<Quelloperand>`

Beschreibung:
`ADC` addiert den Quelloperanden zum Zieloperanden unter Berücksichtigung des Carry-Flags, so dass ein Überlauf beachtet werden kann. Das Ergebnis wird im Zieloperanden abgelegt. Es können 8- und 16-Bit Operanden verarbeitet werden.

Beispiel:

```
STC         ; Carry-Flag setzen
MOV AL,01   ; AL=1
MOV AH,01   ; AH=1
ADC AL,AH   ; AL=AH+AL+C -> 03
```

Operanden	Beispiel: ADC...		
<Register>,<Register>	`AL,BL`		
<Register>,<Speicher>	`AL,[200]`	oder	AL,Quelle
<Speicher>,<Register>	`[200],AL`	oder	Quelle,AL
<Register>,<Konstante>	`AL,07`		
<Speicher>,<Konstante>	`byte ptr [200],7`	oder	Quelle,07

ADD *Add*

Addieren (ohne Carry-Flag)

Syntax: `ADD <Zieloperand>,<Quelloperand>`

Beschreibung:
Addiert den Quelloperanden zum Zieloperanden. Das Ergebnis wird im Zieloperanden gespeichert. Ein eventueller Überlauf wird durch das Carry-Flag signalisiert, dieses wird aber nicht wie bei ADC in die Addition einbezogen. Es können sowohl 8- als auch 16-Bit Operanden verarbeitet werden.

Operanden	Beispiel: ADD...		
<Register>,<Register>	`AX,BX`		
<Register>,<Speicher>	`AL,[200]`	oder	AL,Quelle
<Speicher>,<Register>	`[200],AL`	oder	Quelle,AL
<Register>,<Konstante>	`AL,0ff`	oder	AL,0ffh
<Speicher>,<Konstante>	`byte ptr [200],0a`	oder	Quelle,0ah

AND *And*

Und-Verknüpfung

Syntax: `AND <Zieloperand>,<Quelloperand>`

Beschreibung:

Der Zieloperand wird bitweise mit dem Quelloperanden nach der Und-Wahrheitstabelle verknüpft, das Ergebnis befindet sich danach im Zieloperanden.

Wahrheitstabelle der UND-Verknüpfung:

Eingang1	Eingang2	Ausgang
0	0	0
0	1	0
1	0	0
1	1	1

Operanden	Beispiel: AND...		
<Register>,<Register>	`AL,BL`		
<Register>,<Speicher>	`AL,[300]`	oder	AL,Quelle
<Speicher>,<Register>	`[300],AL`	oder	Quelle,AL
<Register>,<Konstante>	`AL,07`		
<Speicher>,<Konstante>	`byte ptr [300],07`	oder	Quelle,07

CALL *Call the Subroutine Specified in the Operand*

Aufruf einer im Operanden angegebenen Unterroutine

Syntax: `CALL <Zieladresse>`

Beschreibung:
Speichert die Adresse, die unmittelbar nach dem CALL-Befehl folgt auf dem Stack ab und springt in das durch die Zieladresse angegebene Unterprogramm, welches mit dem Befehl RET beendet werden kann um an der auf dem Stack abgelegten Adresse fortzufahren. Es sind vier verschiedene Aufrufarten möglich.

1. Direkter Intra-Segment Aufruf:

Hierbei liegt die Zieladresse, in einem Assemblerprogramm durch ein Sprunglabel oder einen NEAR-Prozedurtyp dargestellt, innerhalb des gleichen Segmentes. Dementsprechend wird nur der Inhalt des IP-Registers auf dem Stack abgelegt, da das CS-Register identisch ist („NEAR-Call“).

2. Indirekter Intra-Segment Aufruf:

Hier ist die Zieladresse in einem 16-Bit Register oder in zwei Speicherzellen (zuerst Low-Byte dann High-Byte) enthalten. Auch hier kann somit nur innerhalb eines Segmentes verzweigt werden („NEAR-Call“).

3. Direkter Inter-Segment Aufruf:

Bei diesem Aufruf befindet sich die Zieladresse, in einem Assemblerprogramm z.B. durch ein Sprunglabel oder eine FAR-Prozedur definiert, in einem anderen Segment. Aus diesem Grund wird sowohl das IP- als auch das CS-Register als Rücksprungadresse für einen RET-Befehl auf dem Stack abgelegt (FAR-Call).

4. Indirekter Inter-Segment Aufruf:

Die Zieladresse befindet sich hier in zwei aufeinanderfolgenden Words (4 Bytes, zuerst Low-Word, dann High-Word) im Arbeitsspeicher.

Operanden	Beispiel:		
<NEAR Adresse>	`call 1b`	oder	call MyLab
<FAR Adresse>		oder	call es:word ptr MyLab
<16-Bit Speicheradresse>	`call [100]`	oder	call word ptr Ziel
<16-Bit Register>	`call AX`		
<32-Bit Speicheradresse>	`call far [200]`	oder	call dword ptr Ziel

CBW *Convert Byte into Word*

Byte vorzeichengerecht in Word umwandeln

Syntax: `CBW`

Beschreibung:
Wandelt das Byte im AL-Register vorzeichengerecht in eine 16-Bit Zahl im AX-Register um. Dazu wird Bit 7 des AL-Registers auf alle Bits des AH-Registers übertragen.

CLC *Clear Carry-Flag*

Carry-Flag löschen

Syntax: `CLC`

Beschreibung:
Löscht das Carry-Flag (C)

CLD *Clear Direction-Flag*

Richtungs-Flag löschen

Syntax: `CLD`

Beschreibung:
Löscht das Richtungs-Flag (D)

CLI *Clear Interrupt-Flag*

Interrupt-Flag löschen

Syntax: `CLI`

Beschreibung:
Löscht das Interrupt-Flag (I) und bewirkt damit, dass der Prozessor keine externen Interrupts mehr zulässt.

CMC *Complement Carry-Flag*

Carry-Flag invertieren

Syntax: `CMC`

Beschreibung:
Kehrt den Inhalt des Carry-Flags (C) um.

CMP *Compare*

Vergleichen

Syntax: `CMP <Operand 1>,<Operand 2>`

Beschreibung:
Vergleicht die angegebenen Operanden und setzt in Abhängigkeit davon bestimmte Flags für z.B. einen bedingten Sprungbefehl (der Vergleich wird intern durch die Subtraktion der beiden Operanden durchgeführt, es wird kein Register beeinflusst).

Operanden	Beispiel: CMP...		
<Register>,<Register>	`BX,CX`		
<Register>,<Speicher>	`AL,[220]`	oder	AL,Zahl
<Speicher>,<Register>	`[220],Zahl`	oder	Zahl,AL
<Register>,<Konstante>	`AH,ff`	oder	AH,255
<Speicher>,<Konstante>	`byte ptr [220],07`	oder	Zahl,07

`CMPS, CMPSB, CMPSW` *Compare Strings*

Strings vergleichen

Syntax: `CMPS` `<Zielstring>,<Quellstring>`
`CMPSB`
`CMPSW`

Beschreibung:
Vergleicht einen Zielstring, der durch die Register ES:DI adressiert wird, mit einem Quellstring, der durch die Register DS:SI adressiert wird. Je nach Wert des Richtungsflags (D) wird in aufsteigender (D=0) oder in absteigender (D=1) Reihenfolge verglichen. Der Befehl CMPSB vergleicht byteweise, der Befehl CMPSW wortweise. Die Register DI und SI dienen als Laufindex und werden dementsprechend um eins oder um zwei erhöht oder erniedrigt. Um mehr als nur ein Element zu vergleichen, kann dem Befehl CMPS ein REP-Präfix (siehe auch REPE bzw. REPZ oder REPNE bzw. REPNZ) vorangestellt werden. Die Anzahl der durchzuführenden Vergleiche muss dann im CX-Register stehen. Anders als bei CMP wird hier intern der Quellstring vom Zielstring subtrahiert, so dass zwangsläufig andere Sprungbefehle eingesetzt werden müssen:

Sprung wenn			mit Vorzeichen	ohne Vorzeichen
Zielstring	>	Quellstring	`JL`	`JB`
Zielstring	>=	Quellstring	`JLE`	`JBE`
Zielstring	<	Quellstring	`JG`	`JA`
Zielstring	<=	Quellstring	`JGE`	`JAE`
Zielstring	=	Quellstring	`JE`	`JE`
Zielstring	<>	Quellstring	`JNE`	`JNE`

`CWD` *Convert Word to Doubleword*

Wort vorzeichengerecht in Doppelwort umwandeln

Syntax: `CWD`

Beschreibung:
Wandelt das Word im AX-Register vorzeichengerecht in eine 32-Bit Zahl im Registerpaar DX:AX um. Dazu wird Bit 15 des AX-Registers in alle Bits des DX-Registers übertragen.

`DAA` *Decimal Adjust for Addition*

Dezimale Anpassung für die Addition

Syntax: `DAA`

Beschreibung:
Wandelt nach einer Addition den Inhalt des AL-Registers in zwei gepackte BCD-Zahlen um. Falls die unteren vier Bits von AL größer als 9 sind, wird AL zunächst um 6 erhöht und das A-Flag gesetzt. Falls die oberen 4 Bits ebenfalls größer als 9 sind, wird AL um 60 erhöht und das C-Flag gesetzt. Durch die gesetzten Flags wird der Überlauf gekennzeichnet.

Beispiele:

```
AL=00010111 Bin (= 17 Hex oder 17 BCD)
AH=01111001 Bin (= 79 Hex oder 79 BCD)
ADD AL,AH -> AL = 90 Hex
aber: 17 BCD + 79 BCD = 96 BCD
DAA       -> AL = 96 Hex = 96 BCD

AL=E0 Hex ( = ?? BCD)
DAA -> AL=40 Hex und C=1
```

DAS *Decimal Adjust for Subtraction*

Dezimale Anpassung für die Subtraktion

Syntax: `DAS`

Beschreibung:
Wandelt nach einer Subtraktion den Inhalt des AL-Registers in zwei gepackte BCD-Zahlen um. Falls die unteren vier Bits von AL größer als 9 sind, wird AL zunächst um 6 erniedrigt und das A-Flag gesetzt. Falls die oberen 4 Bits ebenfalls größer als 9 sind, wird AL um 60 erniedrigt und das C-Flag gesetzt. Durch die gesetzten Flags wird der Überlauf gekennzeichnet. Dieser Befehl stellt somit das Gegenstück zu dem Befehl `DAA` dar.

DEC *Decrement*

Dekrementieren

Syntax: `DEC <Operand>`

Beschreibung:
Der Operand wird um eins erniedrigt.

Operanden	Beispiel: DEC...		
< 8-Bit Register>	`AL`		
<16-Bit Register>	`BX`		
< 8-Bit Speicher>	`byte ptr [0080]`	oder	Zahl
<16-Bit Speicher>	`word ptr [0200]`	oder	Wert

DIV *Divide*

Vorzeichenlose Division

Syntax: `DIV <Quelloperand>`

Beschreibung:
Handelt es sich bei dem Quelloperanden um eine 8-Bit Zahl, wird das AX-Register durch den Quelloperanden geteilt (16-Bit Division). Das Ergebnis steht dann im AL-Register und der Rest im AH-Register. Handelt es sich bei dem Quelloperanden um eine 16-Bit Zahl, wird das Registerpaar DX:AX durch den Quelloperanden geteilt (32-Bit Division). Das Ergebnis steht dann im AX-Register und der Rest im DX-Register. Ergibt sich ein Ergebnis, das größer als 0FF hex (bei der 16-Bit Division) oder größer als 0FFFF Hex (bei der 32-Bit Division) ist, oder ist der Operand gleich Null, wird ein Interrupt 0 ausgelöst, welcher auf eine Unterroutine zeigt, die eine entsprechende Fehlerbehandlung ausführt.

Operanden	Beispiel: DIV...		
< 8-Bit Register>	`BL`		
<16-Bit Register>	`BX`		
< 8-Bit Speicher>	`byte ptr [0200]`	oder	Zahl
<16-Bit Speicher>	`word ptr [0300]`	oder	Wert

ESC *Escape to external Device*

Umschalten zu einem Controller

Syntax: `ESC <Opcode-Konstante>,<Quelloperand>`

Beschreibung:
`ESC` bewirkt, dass der Prozessor den Inhalt einer Speicherstelle zwar liest, aber nicht weiterverarbeitet, sondern diese zu einem Controller (oder einem anderen Baustein) schickt. Der Opcode stellt dabei den Befehl für den Controller (z.B. math. Coprozessor) dar, der Quelloperand enthält die an den Controller weiterzuleitenden Operanden zum Opcode.

Quelloperand	Beispiel: ESC...		
<8-Bit-Register>	`5,AL`		
<8-Bit Speicher>	`5,byte ptr [0200]`	oder	Zahl

HLT *Halt*

Prozessor anhalten

Syntax: `HLT`

Beschreibung:
`HLT` bringt den Prozessor in den Wartezustand. Dieser kann nur durch die Reset-Leitung oder durch einen nicht maskierbaren oder durch einen zwar maskierbaren, aber nicht gesperrten Interrupt aufgehoben werden.

IDIV *Integer Signed Divide*

Division mit Vorzeichen

Syntax: `IDIV <Quelloperand>`

Beschreibung:
Bei diesem Befehl wird das oberste Bit des Divisors und des Dividenden als Vorzeichen interpretiert (Zweierkomplement). Handelt es sich bei dem Quelloperanden um eine 8-Bit Zahl, wird das AX-Register durch den Quelloperanden geteilt (16-Bit Division). Das Ergebnis steht dann im AL-Register und der Rest im AH-Register. Handelt es sich bei dem Quelloperanden um eine 16-Bit Zahl, wird das Registerpaar DX:AX durch den Quelloperanden geteilt (32-Bit Division). Das Ergebnis steht dann im AX-Register und der Rest im DX-Register. Zu beachten ist, dass das Ergebnis im Bereich -128 bis + 127 bei der 8-Bit- und -32768 bis +32767 bei der 16-Bit Division liegen muss. Ist das Ergebnis größer, signalisiert das C-Flag den Überlauf. Ist der Quelloperand gleich Null, wird der Interrupt 0 ausgeführt, welcher auf eine Routine zur Fehlerbehandlung zeigt.

Operanden	Beispiel: IDIV...		
< 8-Bit Register>	`BL`		
<16-Bit Register>	`BX`		
< 8-Bit Speicher>	`byte ptr [0200]`	oder	Zahl
<16-Bit Speicher>	`word ptr [0300]`	oder	Wert

IMUL *Integer Signed Multiplication*

Multiplikation mit Vorzeichen

Syntax: `IMUL <Quelloperand>`

Beschreibung:
Bei diesem Befehl wird das oberste Bit der Operanden als Vorzeichen interpretiert (Zweierkomplement). Ist der Quelloperand ein Byte, wird der Inhalt des AL-Registers mit dem Quelloperanden multipliziert. Das Ergebnis wird im AX-Register gespeichert. Ist der Quelloperand ein Wort, wird das AX-Register mit dem Quelloperanden multipliziert und das Ergebnis im Registerpaar DX:AX gespeichert. Reicht die obere Hälfte (AH bei Byteoperand, DX bei Wortoperand) nicht zur vorzeichengerechten Darstellung des Ergebnisses aus, wird das C- und das O-Flag gesetzt.

Operanden	Beispiel: IMUL...		
< 8-Bit Register>	`BL`		
<16-Bit Register>	`BX`		
< 8-Bit Speicher>	`byte ptr [02E0]`	oder	Zahl
<16-Bit Speicher>	`word ptr [03E0]`	oder	Wert

IN *Input to AL/AX*

Eingabe in AL/AX

Syntax:
```
IN AL,<Konstante von 0 .. 255>
IN AX,<Konstante von 0 .. 255>
IN AL,DX
IN AX,DX
```

Beschreibung:
Liest einen Wert von einem I/O-Port in das AL- bzw. AX-Register ein. Bei diesem Befehl kann wahlweise ein konstanter Wert im Bereich von 0 .. 255 oder das DX-Register angegeben werden, welches die Nummer des einzulesenden Ports enthält (0 .. 65535).

INC *Increment*

Inkrementieren

Syntax: `DEC <Operand>`

Beschreibung:
Der Operand wird um eins erhöht.

Operanden	Beispiel: INC...
< 8-Bit Register>	`AL`

<16-Bit Register>	`BX`	
< 8-Bit Speicher>	`byte ptr [0080]`	oder Zahl
<16-Bit Speicher>	`word ptr [0200]`	oder Wert

`INT` *Interrupt*

Software-Interrupt (Systemaufruf)

Syntax: `INT <Konstante von 0 .. 255>`

Beschreibung:
Das Flagregister, das CS- und das IP-Register werden auf dem Stack abgelegt und es wird zu dem Unterprogramm verzweigt, auf das die angegebene Konstante in der Interrupt-Vektor Tabelle (Adresse 0:4*Konstante) verweist. Über den `IRET`-Befehl kehrt das Unterprogramm zum aufrufenden Programm zurück.

`INT 3` *Breakpoint Interrupt*

Syntax: `INT 3`

Beschreibung:
Stellt einen Sonderfall des `INT`-Befehls dar. Dieser Befehl ist nur ein 1 Byte lang und wird hauptsächlich von Debug-Programmen genutzt.

INTO *Interrupt on Overflow*

Unterbrechung bei Überlauf

Syntax: `INTO`

Beschreibung:
Verursacht dann eine Programmunterbrechung, wenn das O-Flag gesetzt ist. Bei gesetztem Flag wird ein Interrupt 4 ausgeführt, d. h. das Unterprogramm wird aufgerufen, dessen Adresse an der Stelle 0:4*4 in der Interrupt-Vektor-Tabelle hinterlegt ist.

IRET *Interrupt Return*

Rückkehr von Interrupt-Routine

Syntax: `IRET`

Beschreibung:
Der ursprüngliche Inhalt der Register CS, IP und des Flagregisters wird vom Stack aus wieder hergestellt (oberste drei Worte des Stacks) und somit eine mittels des INT-Befehls aufgerufene Interupt-Routine beendet.

JA *Jump on Above*

Sprung, wenn absolut größer

Syntax: `JA <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenloser Zahlen das Ergebnis „Größer“ resultiert. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: C=0 Z=0

Äquivalent: `JNBE`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

JAE *Jump on Above or Equal*

Sprung, wenn absolut größer oder gleich

Syntax: `JAE <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenloser Zahlen das Ergebnis „Größer gleich“ resultiert. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: C=0

Äquivalent: `JNB`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

JB *Jump on Below*

Sprung, wenn kleiner

Syntax: `JB <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenloser Zahlen das Ergebnis „Kleiner“ resultiert. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: C=1

Äquivalent: `JNAE`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JBE` *Jump on Below or Equal*

Sprung, wenn kleiner oder gleich

Syntax: `JBE <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenloser Zahlen das Ergebnis „Kleiner gleich“ resultiert. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: C=1 Z=1

Äquivalent: `JNA`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JC` *Jump if CF=1*

Sprung, wenn Carry-Flag=1

Syntax: `JC <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn das Carry-Flag gesetzt ist. Das Sprungziel muss im Bereich -128 .. +127 liegen.
Status der Flags bei einem Sprung: C=1

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JCXZ` *Jump on CX Zero*

Sprung, wenn CX-Register gleich Null

Syntax: `JCXZ <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn das CX-Register gleich Null ist. Das Sprungziel muss im Bereich -128 .. +127 liegen.

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JE` *Jump on Equal*

Sprung, wenn gleich

Syntax: `JE <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier Zahlen das Ergebnis „Gleich“ resultiert. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen. Der Befehl ist identisch mit dem häufig verwendeten Befehl JZ.
Status der Flags bei einem Sprung: Z=0

Äquivalent: `JZ`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JG` *Jump on Greater*

Sprung, wenn größer

Syntax: `JG <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenbehafteter Zahlen (Zweierkomplement) des Ergebnis „Größer“ resultiert. Das Sprungziel muss im Bereich 128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: S=0 Z=0

Äquivalent: `JNLE`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JGE` *Jump on Greater or Equal*

Sprung, wenn größer gleich

Syntax: `JGE <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenbehafteter Zahlen (Zweierkomplement) des Ergebnis „Größer gleich“ resultiert. Das Sprungziel muss im Bereich 128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: S=0

Äquivalent: `JNL`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JL` *Jump on Less*

Sprung, wenn kleiner

Syntax: `JL <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenbehafteter Zahlen (Zweierkomplement) des Ergebnis „Kleiner“ resultiert. Das Sprungziel muss im Bereich 128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: S ungleich O

Äquivalent: `JNGE`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JLE` *Jump on Less or Equal*

Sprung, wenn kleiner gleich

Syntax: `JLE <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenbehafteter Zahlen (Zweierkomplement) des Ergebnis „Kleiner gleich“ resultiert. Das Sprungziel muss im Bereich 128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: S ungleich O, Z=1

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JMP` *Unconditional Jump*

Unbedingter Sprung

Syntax: `JMP <Adresse>`

Beschreibung:
Springt zur angegebenen Adresse. Es werden vier verschiedene Sprungarten unterschieden:

1. Direkter Intra-Segment Sprung:

Die Zieladresse wird aus dem IP-Register und einem Abstandswert gebildet. Da der Programmierer in aller Regel mit Sprungzielen arbeitet, wird die in der Befehlssyntax beschriebene Adresse für den Programmierer unsichtbar vom Assembler/Debugger in einen Abstandswert umgesetzt. Das maximale Sprungziel liegt damit innerhalb eines 64k-Segmentes. Je nach Größe dieses Abstandswertes wird zwischen einem SHORT-Sprung (1 Byte für den Wert, Sprungweite +/-127 Byte) und einem NEAR-Sprung (2 Byte für den Wert, Sprungweite +/-32767 Byte) unterschieden. Auf diese Weise funktioniert dieser Sprungbefehl sogar bei Programmen, die sich selbst im Arbeitsspeicher verschieben.

2. Indirekter Intra-Segment Sprung:

Die Zieladresse ist in einem 16-Bit Register oder in einem Wort (High-Byte/Low-Byte) im Arbeitsspeicher enthalten (NEAR-Sprung).

3. Direkter Inter-Segment Sprung:

Die Zieladresse besteht aus einem Segment:Offsetanteil, so dass jede Adresse innerhalb des 1 MByte-Adressraums erreicht werden kann (FAR-Sprung).

4. Indirekter Inter-Segment Sprung:

Die Zieladresse steht in einem Doppelwort (4 Bytes) im Arbeitsspeicher (FAR-Sprung).

Operanden	Beispiel:		
<SHORT Adresse>	`jmp short 110`	oder	`jmp short MyLabel`
<NEAR Adresse>	`jmp 1A2`	oder	`jmp MyLabel`
<16-Bit Register>	`jmp AX`		
<16-Bit Speicher>	`jmp [200]`	oder	`jmp word ptr Ziel`
<FAR Adresse>			`jmp es:word ptr Mylabel`
<32-Bit Speicheradresse>	`jmp far [200]`	oder	`jmp dword ptr Ziel`

JNE *Jump on Not Equal*

Sprung, wenn nicht gleich

Syntax: `JNE <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn bei einem Vergleich zweier vorzeichenbehafteter Zahlen (Zweierkomplement) des Ergebnis „Ungleich“ resultiert. Das Sprungziel muss im Bereich 128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: Z=0

Äquivalent: `JNZ`

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JNO` *Jump on Not Overflow*

Sprung, wenn Overflow-Flag nicht gesetzt

Syntax: `JNO <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn das O-Flag nicht gesetzt ist. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: O=0

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JNS` *Jump on Not Sign*

Sprung, wenn Vorzeichen-Flag nicht gesetzt

Syntax: `JNS <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn das S-Flag nicht gesetzt ist. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: S=0

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JO` *Jump on Overflow*

Sprung, wenn Overflow-Flag gesetzt

Syntax: `JO <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn das O-Flag gesetzt ist. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: O=1

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JP` *Jump on Parity*

Sprung, wenn Paritäts-Flag gesetzt

Syntax: `JP <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn das P-Flag gesetzt ist. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: P=1

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

`JS` *Jump on Sign*

Sprung, wenn Sign-Flag gesetzt

Syntax: `JS <Adresse>`

Beschreibung:
Bewirkt einen Sprung zur angegebenen Adresse, wenn das S-Flag gesetzt ist. Das Sprungziel muss im Bereich -128 .. +127 Bytes liegen.
Status der Flags bei einem Sprung: S=1

Bemerkung:
In der internen Binärcodierung des Sprungbefehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

LAHF *Load Flags into AH-Register*

Lade Flags in das AH-Register

Syntax: `LAHF`

Beschreibung:
Lädt die untere Hälfte des Flag-Registers in das AH-Register

LDS *Load Register and DS from Memory*

Lade Register und DS-Register aus dem Speicher

Syntax: `LDS <Zielregister>,<Adresse>`

Beschreibung:
`LDS` lädt ein unter der angegebenen Adresse (Label) hinterlegtes Doppelwort in das angegebene Register (erstes Wort) und in das DS-Register (zweites Wort). In der Regel handelt es sich bei dem Doppelwort um Segment- und Offsetadresse, welche dann mit einem einzigen Befehl in die Register geladen werden können (z.B. bei den Stringbefehlen `LODS` und `STOS`).

LEA *Load Effective Adress*

Effektive (Offset-) Adresse laden

Syntax: `LEA <Zielregister>,<Adresse>`

Beschreibung:
`LEA` lädt die Offsetadresse einer Speicherstelle (und nicht deren Inhalt!) in das angegebene Register. Während `MOV AX,[200]` den Wert, den die Speicherstelle 200 enthält, in das AX-Register überträgt, bewirkt `LEA AX,[200]`, dass der Wert 200, welcher der Speicheradresse entspricht, in das AX-Register geladen wird. Dieser Befehl ermöglicht es, Indexregister mit Tabellen-Anfangsadressen zu laden,
z.B. `LEA BX,[100] ... MOV AX,[BX]`. Der Befehl ist v.a. bei symbolischen Assemblern interessant.

LES *Load Register and ES from Memory*

Lade Register und ES-Register aus dem Speicher

Syntax: `LES <Zielregister>,<Adresse>`

Beschreibung:
LES lädt ein unter der angegebenen Adresse (Label) hinterlegtes Doppelwort in das angegebene Register (erstes Word) und in das ES-Register (zweites Word). In der Regel handelt es sich bei dem Doppelwort um Segment- und Offsetadresse, welche dann mit einem einzigen Befehl in die Register geladen werden können (z.B. bei den Stringbefehlen LODS und STOS).

LOCK *Bus Lock Signal*

Bus sperren

Syntax: `LOCK <Maschinen-Befehl>`

Beschreibung:
Bei `LOCK` handelt es sich um ein Präfix, welches jedem Maschinenbefehl vorangestellt werden kann. Durch dieses Präfix wird der Bus solange gesperrt, bis die Abarbeitung des angegebenen Maschinenbefehls beendet ist. Dieser Befehl wird z.B. benötigt, um sicherzustellen, dass auf den Arbeitsspeicher nicht mehr als ein Prozessor zugreifen kann.

LODSB, LODSW *Load Byte/Word into AL/AX*

Bytes/Words in AL/AX laden

Syntax: `LODSB`
`LODSW`

Beschreibung:
Lädt das Byte (`LODSB`) bzw. Word (`LODSW`), das durch die Register DS:SI adressiert wird, in das AL- bzw. AX Register. Danach wird das SI-Register in Abhängigkeit des Richtungsflags D um eins bei dem Befehl `LODSB` und um zwei bei dem `LODSW` erhöht (D=0) bzw. erniedrigt (D=1). `LODSB/LODSW` kann in Verbindung mit dem Wiederholungspräfix `REP` eingesetzt werden. Als Zähler für die Wiederholungen dient dann das CX-Register. Allerdings wird AL/AX überschrieben.

LOOP *Decrement CX and Jump on Not Zero*

CX erniedrigen und Sprung, solange CX ungleich 0

Syntax: `LOOP <Adresse>`

Beschreibung:
`LOOP` erniedrigt das CX-Register, welches als Zähler dient, und springt, solange CX ungleich Null ist, zur angegebenen Adresse. CX wird zuerst erniedrigt, bevor die Prüfung auf Null stattfindet. Das Sprungziel muss innerhalb eines Bereiches von -128 .. +127 Bytes liegen.

Bemerkung:

In der internen Binärcodierung des LOOP-Befehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

LOOP *Decrement CX and Jump on Not Zero and Z=1*

CX erniedrigen und Sprung, solange CX ungleich 0 und Z=1

Syntax: LOOPE <Adresse>

Beschreibung:
LOOPE erniedrigt das CX-Register, welches als Zähler dient, und springt, solange CX ungleich Null und das Z-Flag gesetzt ist, zur angegebenen Adresse. CX wird zuerst erniedrigt, bevor die Prüfung auf Null und des Z-Flags stattfindet. Ein Sprung erfolgt nur, wenn beide Bedingungen erfüllt sind. Das Sprungziel muss innerhalb eines Bereiches von -128 .. +127 Bytes liegen.

Äquivalent: LOOPZ

Bemerkung:
In der internen Binärcodierung des LOOPE-Befehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

LOOPNE *Decrement CX and Jump on Not Zero and Z=0*

CX erniedrigen und Sprung, solange CX ungleich 0 und Z=0

Syntax: LOOPNE <Adresse>

Beschreibung:
LOOPNE erniedrigt das CX-Register, welches als Zähler dient, und springt, solange CX ungleich Null und das Z-Flag zurückgesetzt ist, zur angegebenen Adresse. CX wird zuerst erniedrigt, bevor die Prüfung auf Null und des Z-Flags stattfindet. Ein Sprung erfolgt nur, wenn beide Bedingungen erfüllt sind. Das Sprungziel muss innerhalb eines Bereiches von -128 .. +127 Bytes liegen.

Äquivalent: LOOPNZ

Bemerkung:
In der internen Binärcodierung des LOOPNZ-Befehls wird nicht die Zieladresse gespeichert, sondern ein Abstandswert, der sich immer auf das momentane IP-Register bezieht.

MOV *Move Data*

Daten übertragen

Syntax: MOV <Ziel>,<Quelle>

Beschreibung:
Der Quelloperand wird in den Zieloperanden kopiert.

Operanden		Beispiel: MOV...		
<Register	>,<Register >	ax,bx		
<Speicher	>,<Register >	byte ptr [110],bl	oder	Zahl,bl
<Register	>,<Speicher >	bx,word ptr [200]	oder	bx,Wert
<Speicher	>,<Konstante >	byte ptr [200],77	oder	Zahl,77
<Register	>,<Konstante >	al,5		
<SegRegister	>,<Speicher >	ds,word ptr [200]	oder	ds,Wert
<Speicher	>,<SegRegister >	word ptr [200],ds	oder	Wert,ds
<SegRegister	>,<Register >	ds,ax		
<Register	>,<SegRegister >	bx,es		

MOVSB, MOVSW *Move Byte/Word from Memory to Memory*

Byte/Word von Speicher zu Speicher übertragen

Syntax: MOVSB
MOVSW

Beschreibung:
Überträgt den Inhalt der durch das Registerpaar DS:SI adressierten Speicherstelle in die Speicherstelle, die durch das Registerpaar ES:DI adressiert wird. Danach werden das SI-Register und das DI-Register in Abhängigkeit des Richtungsflags DF um eins bei dem Befehl MOVSB und um zwei bei dem Befehl MOVSW erhöht (D=0) bzw. erniedrigt (D=1). MOVSB/MOVSW kann in Verbindung mit dem Wiederholungspräfix REP eingesetzt werden. Auf diese Weise können Speicherblöcke mit einer Größe bis zu 64 Kilobyte sehr schnell verschoben werden. Als Zähler für die Wiederholungen dient dann das CX-Register.

MUL *Multiply*

Multiplikation (ohne Vorzeichen)

Syntax: MUL <Quelloperand>

Beschreibung:
Ist der Quelloperand ein Byte, wird der Inhalt des AL-Registers mit dem Quelloperanden multipliziert. Das Ergebnis wird im AX-Register gespeichert. Ist der Quelloperand ein Wort, wird das AX-Register mit dem Quelloperanden multipliziert und das Ergebnis im Registerpaar

DX:AX gespeichert. Ist die obere Hälfte (AH bei Byteoperand, DX bei Wortoperand) nach erfolgter Multiplikation ungleich Null, wird das C-Flag und das O-Flag gesetzt.

Operanden	Beispiel: MUL...	
< 8-Bit Register>	`BL`	
<16-Bit Register>	`BX`	
< 8-Bit Speicher>	`byte ptr [02E0]`	oder Zahl
<16-Bit Speicher>	`word ptr [03E0]`	oder Wert

NEG *Negate*

Negieren

Syntax: `NEG <Operand>`

Beschreibung:
Subtrahiert den angegebenen Operanden von Null und bildet somit dessen Zweierkomplement. Das S-Flag und das C-Flag wird gesetzt, wenn der Operand positiv ist. Ist das Ergebnis Null, wird das Z-Flag gesetzt. Das O-Flag wird gesetzt, wenn der Operand -128 (kleinstes negatives Byte) oder -32768 (kleinstes negatives Wort) ist, da in diesem Fall die Zweierkomplementdarstellung nicht mehr möglich ist.

Operanden	Beispiel: NEG...	
< 8-Bit Register>	`BL`	
<16-Bit Register>	`BX`	
< 8-Bit Speicher>	`byte ptr [03E0]`	oder Zahl
<16-Bit Speicher>	`word ptr [03E0]`	oder Wert

NOP *No Operation*

Keine Operation

Syntax: `NOP`

Beschreibung:
Dieser Befehl führt keine Operation aus und wird z.B. für Verzögerungen eingesetzt.

NOT *Not*

Nicht-Verknüpfung

Syntax: `NOT <Operand>`

Beschreibung:
Alle Bits des angegebenen Operanden werden invertiert. Das Invertieren jedes einzelnen Bits einer Zahl entspricht der Bildung des Einerkomplements.

Operanden	Beispiel: NOT...	
< 8-Bit Register>	`BL`	
<16-Bit Register>	`BX`	
< 8-Bit Speicher>	`byte ptr [03E0]`	oder Zahl
<16-Bit Speicher>	`word ptr [03E0]`	oder Wert

OR *Or*

Oder-Verknüpfung

Syntax: `OR <Zieloperand>,<Quelloperand>`

Beschreibung:
Der Zieloperand wird bitweise mit dem Quelloperanden nach der OR-Wahrheitstabelle verknüpft, das Ergebnis befindet sich danach im Zieloperanden.

Wahrheitstabelle der OR-Verknüpfung:

Eingang1	Eingang2	Ausgang
0	0	0
0	1	1
1	0	1
1	1	1

Operanden	Beispiel: OR...		
<Register>,<Register>	`AL,BL`		
<Register>,<Speicher>	`AL,[300]`	oder	AL,Quelle
<Speicher>,<Register>	`[300],AL`	oder	Quelle,AL
<Register>,<Konstante>	`AL,07`		

<Speicher>,<Konstante> `byte ptr [300],07` oder Quelle,07

`OUT` *Output from AL/AX*

Ausgabe von AL/AX

Syntax:
```
OUT <Konstante von 0 .. 255>,AL
OUT <Konstante von 0 .. 255>,AL
OUT DX,AL
OUT DX,AX
```

Beschreibung:
Gibt den Inhalt des AL- bzw. AX-Registers auf einen I/O-Port aus. Bei diesem Befehl kann wahlweise ein konstanter Wert (Portadresse) im Bereich von 0 .. 255 oder das DX-Register angegeben werden, welches die Adresse des Ports enthält (0 ...65535), auf den geschrieben werden soll.

`POP` *Read Word from Stack*

Wort vom Stack holen

Syntax: `POP <Operand>`

Beschreibung:
`POP` lädt den angegebenen Operanden mit dem obersten Wort des Stacks und erhöht den Stackzeiger (SP-Register) um zwei. Die Werte sind nach der Intel-Konvention (zuerst Low-Byte, dann High-Byte) auf dem Stack gemäß dem LIFO-Prinzip (last in first out) abgespeichert.

Operanden	Beispiel:
<Register >	`POP AX`
<SegRegister >	`POP DS`
<16-Bit Speicher >	`POP [200]` oder POP Wert

`POPF` *Read from Top of Stack into Flag-Register*

Wort vom Stack in das Flag-Register laden

Syntax: `POPF`

Beschreibung:
`POPF` lädt das Flag-Register mit dem obersten Word vom Stack und erhöht den Stackzeiger (SP-Register) um zwei.

Sonderanwendung:
Mit einem bestimmten Wert, der zuvor mit PUSH auf den Stack geschrieben wurde, können alle Flags gleichzeitig gesetzt/zurückgesetzt werden. Die Werte werden nach der Intel-Konvention (zuerst Low-Byte, dann High-Byte) gemäß dem LIFO-Prinzip auf dem Stack abgespeichert.

PUSH *Write to the Top of Stack*

Wort auf dem Stack ablegen

Syntax: PUSH <Operand>

Beschreibung:
PUSH speichert den angegebenen Operanden auf dem Stack ab und erniedrigt den Stackzeiger (SP-Register) um zwei. Die Werte werden nach der Intel-Konvention (zuerst Low-Byte, dann High-Byte) auf dem Stack gemäß dem LIFO-Prinzip (last in first out) abgespeichert.

Operanden	Beispiel:
<Register >	PUSH AX
<Segment-Register >	PUSH CS
<16-Bit Speicher >	PUSH [200] oder PUSH Wert

PUSHF *Write Flag-Register to the Top of Stack*

Flag-Register auf dem Stack ablegen

Syntax: PUSHF

Beschreibung:
PUSHF speichert das Flag-Register auf dem Stack ab und erniedrigt den Stackzeiger (SP-Register) um zwei. Der Wert wird nach der Intel-Konvention (zuerst Low-Byte, dann High-Byte) auf dem Stack gemäß dem LIFO-Prinzip (last in first out) abgespeichert.

RCL *Rotate with Carry-Flag Left*

Mit Carry-Flag links rotieren

Syntax: RCL <Operand>,1
RCL <Operand>,CL

Beschreibung:
Der angegebene Operand wird entweder um 1, oder um die Anzahl entsprechend dem Wert im CL-Register unter Einbeziehung des Carry-Flags nach links rotiert. Dabei wird der aktuelle Zustand des C-Flags in das unterste Bit des Operanden übertragen während alle Bits um eine Stelle weiter nach links rücken. Das oberste Bit des Operanden, für welches kein Platz mehr bleibt, wird dann in das C-Flag übetragen. Das O-Flag wird gesetzt, wenn sich bei der Rotation das Vorzeichen ändert.

Operanden	Beispiel: RCL...		
<Register>	BX,1		
	BX,CL		
<Speicher>	byte ptr [200],1	oder	Zahl,1
	word ptr [300],CL	oder	Wert,CL

RCR *Rotate with Carry-Flag Right*

Mit Carry-Flag rechts rotieren

Syntax: RCR <Operand>,1
RCR <Operand>,CL

Beschreibung:
Der angegebene Operand wird entweder um 1, oder um die Anzahl entsprechend dem Wert im CL-Register unter Einbeziehung des Carry-Flags nach rechts rotiert. Dabei wird der aktuelle Zustand des C-Flags in das oberste Bit des Operanden übertragen während alle Bits um eine Stelle weiter nach rechts rücken. Das unterste Bit des Operanden, für welches kein Platz mehr bleibt, wird dann in das C-Flag übertragen. Das O-Flag wird gesetzt, wenn sich bei der Rotation um Eins das Vorzeichen ändert.

Operanden	Beispiel: RCR...		
<Register>	`AX,CL`		
	`BX,1`		
<Speicher>	`byte ptr [200],1`	oder	Zahl,1
	`word ptr [300],CL`	oder	Wert,CL

REP *Repeat Following String Instruction until CX=0*

Wiederholung des folgenden String Befehls

Syntax: `REP <Stringbefehl>`

Beschreibung:
Der angegebene Stringbefehl wird entsprechend dem Wert im CX-Register wiederholt. Nach jeder Wiederholung wird das CX-Register um 1 erniedrigt. Die Wiederholung endet, wenn CX gleich Null ist. Derartig wiederholte Stringbefehle können durch einen Interrupt unterbrochen werden. Gültige Stringbefehle sind:
`MOVS`, `MOVSB`, `MOVSW`, `STOSB` und `STOSW`

REPE *Repeat Following String Instruction until CX<>0 and Z=1*

Wiederholung des folgenden String Befehls solange CX<>0 und Z=1

Syntax: `REPE <Stringbefehl>`

Beschreibung:
Der angegebene Stringbefehl wird entsprechend dem Wert im CX-Register wiederholt, solange CX ungleich Null und das Z-Flag gesetzt ist. Nach jeder Wiederholung wird das CX-Register um 1 erniedrigt. Derartig wiederholte Stringbefehle können durch einen Interrupt unterbrochen werden. Sinnvolle Stringbefehle sind:
`CMPS`, `CMPSB`, `CMPSW` und `SCAS`

Äquivalent: `REPZ`

REPNE *Repeat Following String Instruction until CX<>0 and Z=0*

Wiederholung des folgenden String Befehls solange CX<>0 und Z=0

Syntax: `REPNE <Stringbefehl>`

Beschreibung:
Der angegebene Stringbefehl wird entsprechend dem Werte im CX-Register wiederholt, solange CX ungleich Null und das Z-Flag gelöscht ist. Nach jeder Wiederholung wird das CX-Register um 1 erniedrigt. Derartig wiederholte Stringbefehle können durch einen Interrupt unterbrochen werden. Sinnvolle Stringbefehle sind:
`CMPS`, `CMPSB`, `CMPSW` und `SCAS`

Äquivalent: `REPNZ`

RET *Return from Near-Subroutine*

Rückkehr von Near-Unterprogramm

Syntax: `RET`

Beschreibung:
Der ursprüngliche Inhalt des IP-Registers wird vom Stack aus wiederhergestellt (oberstes Wort auf dem Stack) und somit das mittels (Near-) CALL aufgerufene Unterprogramm beendet.

RETF *Return from Far-Subroutine*

Rückkehr von Far-Unterprogramm

Syntax: `RETF`

Beschreibung:
Der ursprüngliche Inhalt des IP- und des CS-Registers wird vom Stack aus wieder hergestellt (obere zwei Worte auf dem Stack) und somit das mittels eines Far-CALLs aufgerufene Unterprogramm beendet.

ROL *Rotate Left*

Links rotieren

Syntax:
```
ROL <Operand>,1
ROL <Operand>,CL
```

Beschreibung:
Der angegebene Operand wird entweder um 1, oder um die Anzahl entsprechend dem Wert im CL-Register nach links rotiert. Alle Bits rücken um n Positionen nach links während die obersten Bits direkt in die untersten Bits des Operanden übertragen werden. Das O-Flag wird gesetzt, wenn sich bei der Rotation das Vorzeichen ändert.

Operanden	Beispiel: ROL...
<Register>	`AX,CL`
	`BX,1`
<Speicher>	`byte ptr [200],1` oder Zahl,1
	`word ptr [300],CL` oder Wert,CL

ROR *Rotate Right*

Rechts rotieren

Syntax:
```
ROR <Operand>,1
ROR <Operand>,CL
```

Beschreibung:
Der angegebene Operand wird entweder um 1, oder um die Anzahl entsprechend dem Wert im CL-Register nach rechts rotiert. Ale Bits rücken um n Positionen nach rechts während die untersten Bits direkt in die obersten Bits des Operanden übertragen werden. Das O-Flag wird gesetzt, wenn sich bei der Rotation das Vorzeichen ändert.

Operanden	Beispiel: ROR...
<Register>	`AX,1`
	`BX,CL`
<Speicher>	`byte ptr [200],1` oder Zahl,1
	`word ptr [300],CL` oder Wert,CL

SAHF *Store the AH-Register into Flags*

AH-Register in unteres Byte des Flag-Registers laden

Syntax: `SAHF`

Beschreibung:
Die unteren 8 Bit des Flag-Registers werden entsprechend dem Inhalt des AH-Registers gesetzt.

SHL *Shift Left*

Links schieben

Syntax: `SHL <Operand>,1`
`SHL <Operand>,CL`

Beschreibung:
Der angegebene Operand wird entweder um 1, oder um die Anzahl entsprechend dem Wert im CL-Register nach links geschoben. Alle Bits rücken um n Positionen nach links während das zuletzt hinausgeschobene Bit in das C-Flag übertragen und die untersten Bits mit Nullen aufgefüllt werden. Das O-Flag wird gesetzt, wenn sich bei der Verschiebung das Vorzeichen ändert.

Äquivalent: `SAL`

Operanden	Beispiel: SHL...		
<Register>	`AX,1`		
	`BX,CL`		
<Speicher>	`byte ptr [200],1`	oder	Zahl,1
	`word ptr [300],CL`	oder	Wert,CL

SAR *Shift arithmetic Right*

Arithmetisch nach Rechts schieben

Syntax: `SAR <Operand>,1`
`SAR <Operand>,CL`

Beschreibung:
Alle Bits des angegebenen Operanden werden entweder um 1, oder um die Anzahl entsprechend dem Wert im CL-Register nach rechts geschoben. Alle Bits rücken um n Positionen nach rechts, während das unterste Bit in das C-Flag übertragen wird. Da das MSB durch diese Verschiebung nicht beeinflusst wird, bleibt das Vorzeichen des Operanden erhalten. Das O-Flag wird dementsprechend nicht verändert.

Operanden	Beispiel: SAL...		
<Register>	`AX,CL`		
	`BX,1`		
<Speicher>	`byte ptr [200],1`	oder	Zahl,1
	`word ptr [300],CL`	oder	Word,CL

SBB *Subtract with Borrow*

Subtrahieren mit Übertrag

Syntax: `SBB <Zieloperand>,<Quelloperand>`

Beschreibung:
Subtrahiert den Quelloperanden vom Zieloperanden unter Berücksichtigung des Carry-Flags, so dass ein Übertrag berücksichtigt werden kann. Das Ergebnis wird im Zieloperanden abgelegt. Es können 8- und 16-Bit Operanden verarbeitet werden.

Beispiel:

```
STC        ; Carry-Flag setzen
MOV AL,02  ; AL=2
MOV AH,01  ; AH=1
SBB AL,AH  ; -> AL=0
```

Operanden	Beispiel: SBB...		
<Register>,<Register>	`AX,BX`		
<Register>,<Speicher>	`AL,[200]`	oder	AL,Quelle
<Speicher>,<Register>	`[200],AL`	oder	Quelle,AL
<Register>,<Konstante>	`AL,ff`	oder	AL,0ffh
<Speicher>,<Konstante>	`byte ptr [200],0a`	oder	Quelle,0ah

SCASB, SCASW *Compare Byte/Word with AL/AX*

Durchsuchen eines Strings nach dem Inhalt von AL/AX

Syntax: `SCASB`
`SCASW`

Beschreibung:
Durchsucht den Inhalt des Strings, der durch das Registerpaar ES:DI adressiert wird, nach dem Inhalt des AL- (`SCASB`) bzw. des AX-Registers (`SCASW`). Das DI-Register wird in Abhängigkeit des Richtungsflags D um eins bei dem Befehl `SCASB` und um zwei bei dem Befehl `SCASW` erhöht (D=0) bzw. erniedrigt (D=1). `SCASB`/ `SCASW` kann in Verbindung mit dem Wiederholungspräfixen `REPE` und `REPNE` eingesetzt werden. Der String lässt sich dann solange durchsuchen, bis das CX-Register auf Null heruntergezählt bzw. das Z-Flag seinen Zustand verändert.

SHR *Shift Right*

Rechts schieben

Syntax: `SHR <Operand>,1`
`SHR <Operand>,CL`

Beschreibung:
Der angegebene Operand wird entweder um 1, oder um die Anzahl entsprechend dem Wert im CL-Register nach rechts geschoben. Alle Bits rücken um n Positionen nach rechts während das zuletzt heraus geschobene Bit in das C-Flag übertragen und das oberste Bit mit einer Null aufgefüllt wird. Das O-Flag wird gesetzt, wenn sich das Vorzeichen ändert.

Operanden	Beispiel: SHR...		
<Register>	`AX,1`		
	`BX,CL`		
<Speicher>	`byte ptr [200],1`	oder	Zahl,1
	`word ptr [300],CL`	oder	Wert,CL

STC *Set Carry-Flag*

Setzen des Carry-Flags

Syntax: `STC`

Beschreibung:
Mit diesem Befehl wird das Carry-Flag auf 1 gesetzt.

STD *Set Direction-Flag*

Setzen des Richtungs-Flags

Syntax: `STD`

Beschreibung:
Mit diesem Befehl wird das Richtungs-Flag auf 1 gesetzt.

STI *Set Interrupt-Flag*

Setzen des Interrupt-Flags

Syntax: `STI`

Beschreibung:
Mit diesem Befehl wird das Interrupt-Flag auf 1 gesetzt. Dadurch wird ermöglicht, dass der Prozessor auf externe Brechungen reagieren kann.

STOSB, STOSW *Store AL/AX into Byte/Word*

AL/AX in Byte/Word schreiben

Syntax: `STOSB`
`STOSW`

Beschreibung:

Speichert das AL-Register (STOSB) bzw. das AX-Register (STOSW) in das Byte bzw. Wort, das durch das Registerpaar ES:DI adressiert wird. Danach wird das DI-Register in Abhängigkeit des Richtungsflags D um eins bei dem Befehl STOSB und um zwei bei dem STOSW erhöht (D=0) bzw. erniedrigt (D=1). STOSB/STOSW kann in Verbindung mit dem Wiederholungspräfix REP eingesetzt werden. Als Zähler für die Wiederholungen dient dann das CX-Register.

SUB *Subtract*

Subtrahieren (ohne Carry-Flag)

Syntax: SUB <Zieloperand>,<Quelloperand>

Beschreibung:
Subtrahiert den Quelloperanden vom Zieloperanden. Das Ergebnis wird im Zieloperanden gespeichert. Ein eventueller Überlauf wird durch das Carry-Flag signalisiert, dieses wird aber nicht wie bei SBB in die Subtraktion einbezogen. Es können sowohl 8- als auch 16-Bit Operanden verarbeitet werden.

Operanden	Beispiel: SUB...		
<Register>,<Register>	AX,BX		
<Register>,<Speicher>	AL,[200]	oder	AL,Quelle
<Speicher>,<Register>	[200],AL	oder	Quelle,AL
<Register>,<Konstante>	AL,ff	oder	AL,0ffh
<Speicher>,<Konstante>	byte ptr [200],0a	oder	Quelle,0ah

TEST *Test*

Vergleich durch logisches UND

Syntax: TEST <Zieloperand>,<Quelloperand>

Beschreibung:
TEST führt zwischen Zieloperand und Quelloperand eine UND-Verknüpfung durch. Beide Operanden bleiben dabei unverändert, lediglich die Flags werden beeinflusst. Bei der Ausführung von TEST wird das C-Flag und das O-Flag gelöscht. Damit kann ein folgender Sprung davon abhängig gemacht werden, ob bestimmte Bits im Zieloperanden gesetzt sind.

Operanden	Beispiel: TEST...		
<Register>,<Register>	AX,BX		
<Register>,<Speicher>	AL,[200]	oder	AL,Quelle
<Speicher>,<Register>	[200],AL	oder	Quelle,AL
<Register>,<Konstante>	AL,ff	oder	AL,0ffh
<Speicher>,<Konstante>	byte ptr [200],0a	oder	Quelle,0ah

WAIT *Wait for Assert Signal on Test-Pin*

Warten auf Signal am Test-Pin

Syntax: WAIT

Beschreibung:
Durch WAIT verharrt der Prozessor solange im Wartezustand (5 Taktzyklen), bis ein Signal an der Testleitung anliegt oder ein Interrupt auftritt. Nach Abarbeitung des Interrupts wird der Wartezustand allerdings fortgesetzt.

XCHG *Exchange*

Vertauschen

Syntax: XCHG <Operand 1>,<Operand 2>

Beschreibung:
XCHG vertauscht die beiden Operanden miteinander.

Operanden	Beispiel: XCHG...		
<Register>,<Register>	AX,BX		
<Register>,<Speicher>	AL,[200]	oder	AL,Quelle
<Speicher>,<Register>	[200],AL	oder	Quelle,AL

XLAT *Translate Byte to AL*

Übersetzen

Syntax: XLAT

Beschreibung:
Lädt das Byte, das durch die Addition von BX mit AL als Offsetadresse und dem Register DS als Segmentadresse adressiert wird, in das AL Register.

Beispiel:
An der Stelle DS:200 stehen die Bytes 0A, 0B und 0C. Dann lädt die Befehlssequenz

```
MOV BX,200
MOV AL,01
XLAT
```

den Wert 0B (= Adresse DS:201) in das AL-Register.

XOR *Exclusive OR*

Exclusive ODER-Verknüpfung (XOR)

Syntax: `XOR <Zieloperand>,<Quelloperand>`

Beschreibung:
Der Zieloperand wird bitweise mit dem Quelloperanden nach der XOR-Wahrheitstabelle verknüpft, das Ergebnis befindet sich danach im Zieloperanden.

Wahrheitstabelle der XOR-Verknüpfung:

Eingang1	Eingang2	Ausgang
0	0	0
0	1	1
1	0	1
1	1	0

Operanden	Beispiel: OR...		
<Register>,<Register>	`AL,BL`		
<Register>,<Speicher>	`AL,[300]`	oder	AL,Quelle
<Speicher>,<Register>	`[300],AL`	oder	Quelle,AL
<Register>,<Konstante>	`AL,07`		
<Speicher>,<Konstante>	`byte ptr [300],07`	oder	Quelle,07