

## 2 Einführung in den 8086-Befehlssatz

In diesem Kapitel werden die Befehle des 8086-/8088-Prozessors vorgestellt. Diese CPU's bieten den Befehlssatz der auch bei 80286-, 386- und 80486-Prozessoren im *Real Mode* unter DOS zur Verfügung steht. Die Übungen im Text wurden bewußt so gehalten, daß sie sich mit dem DOS-Debugger (DEBUG.COM) nachvollziehen lassen. Auf den Einsatz eines Assemblers wurde verzichtet, um den Leser nicht durch die (für den Assembler) erforderlichen Steueranweisungen zu verwirren. Um die Programme zusätzlich zu vereinfachen, beschränken wir uns in diesem Kapitel auf COM-Dateien. Wer trotzdem einen Assembler verwenden möchte, dem steht dies frei. Im folgenden Kapitel wird zum Beispiel auf die Erstellung von Assemblerprogrammen mit dem A86-Assembler eingegangen. Für Besitzer von TASM oder MASM bieten spätere Kapitel genügend Anregungen für eigene Experimente.

Bevor jedoch die ersten Assemblerbefehle vorgestellt werden, möchte ich auf die Architektur der 8086-Prozessoren eingehen. Das Verständnis über die Aufteilung des Speichers, die Adressberechnung mittels Segment-Offset und die Bedeutung der Register ist essentiell für die Programmierung in Maschinensprache.

### 2.1 Einführung in die 8086-Architektur

Mit der Vorstellung der 8086-CPU begann INTEL im Jahre 1978 mit der Einführung einer sehr erfolgreichen Prozessorfamilie. Um die damals aus der 8-Bit-Welt vorhandenen Peripheriebausteine mit verwenden zu können, wurde bald eine reduzierte Version des 8086 unter der Typenbezeichnung 8088 angeboten. Die CPU ist bis auf die Buseinheit (Anschaltung der Daten- und Adreßleitungen) funktional mit dem 8086-Prozessor gleich. Im Abstand von einigen Jahren folgten dann leistungsstärkere Prozessoren der Reihe 80286, 80386 und 80486. Allen CPU's ist gemeinsam, daß sie sich in einem zum 8086-Prozessor kompatiblen Modus (Real Mode) betreiben lassen. Dies ist der Modus, den das Betriebssystem MS-DOS (PC-DOS) benutzt. Dies bedeutet: egal welcher Prozessor in Ihrem PC steckt, für den Einstieg in die Assemblerprogrammierung unter MS-DOS genügt die Kenntnis des 8086-Befehlssatzes. Alle nachfolgende Ausführungen beziehen sich deshalb auf das 8086-Modell.

Der Programmierer sieht von der CPU nur die Register. Dies sind interne Speicherstellen, in denen Daten und Ergebnisse für die Bearbeitung abgelegt werden. Für alle im *8086-Real Mode* betriebenen CPUs gilt die in Bild 2.1 gezeigte Registerstruktur.

Alle Register besitzen standardmäßig eine Breite von 16 Bit. Allerdings gibt es noch eine Besonderheit. Die Universalregister AX bis DX sind für arithmetische und logische Operationen ausgebildet und lassen sich deshalb auch als 8 Register zu je 8 Bit aufteilen. Der Endbuchstabe gibt dabei an, um welchen Registertyp es sich handelt. Ein X (z.B. AX) in der Bezeichnung markiert ein 16-Bit-Register. Mit H wird das High Byte und mit L das Low Byte des jeweiligen 16-Bit-Registers als 8-Bit-Register selektiert. Die oberen 8 Bit des AX-Registers werden damit als AH bezeichnet. AL gibt die unteren 8 Bit des Registers AX an. Das gleiche gilt für die anderen drei Register BX, CX und DX. Welche der Register ein Programm verwendet, hängt von den jeweils benutzten Befehlen ab.

AX	AH	AL	SP
BX	BH	BL	BP
CX	CH	CL	SI
DX	DH	DL	DI
	IP		CS
	Flags		DS
			SS
			ES

Bild 2.1: Die 8086-Registerstruktur

Bei der Bearbeitung verschiedener Befehle besitzen die Register eine besondere Bedeutung, die nachfolgend kurz beschrieben wird.

### 2.1.1 Die Universalregister

Die erste Gruppe bilden die vier Universalregister AX, BX, CX und DX.

#### Der Akkumulator (AX)

Der *Akkumulator* lässt sich in zwei 8-Bit-Register (AH und AL) aufteilen, oder als 16-Bit-Register AX benutzen. Dieses Register wird zur Abwicklung von 16-Bit-Multiplikationen und -Divisionen verwendet. Zusätzlich wird es bei den 16-Bit I/O-Operationen gebraucht. Für 8-Bit-Multiplikations- und Divisionsbefehle dienen die 8-Bit-Register AH und AL. Befehle zur dezimalen Arithmetik, sowie die Translate-Operationen benutzen das Register AL.

### Das Base-Register (BX)

Dieses Register läßt sich bei Speicherzugriffen als Zeiger zur Berechnung der Adresse verwenden. Das gleiche gilt für die Translate-Befehle, wo Bytes mit Hilfe von Tabellen umkodiert werden. Eine Unterteilung in zwei 8-Bit-Register (BH und BL) ist möglich. Weitere Informationen finden sich bei der Beschreibung der Befehle, die sich auf dieses Register beziehen.

### Das Count-Register (CX)

Bei Schleifen und Zählern dient dieses Register zur Aufnahme des Zählers. Der LOOP-Befehl wird dann zum Beispiel solange ausgeführt, bis das Register CX den Wert 0 aufweist. Weiterhin ist bei String-Befehlen die Länge des zu bearbeitenden Textbereiches in diesem Register abzulegen. Bei den Schiebe- und Rotate-Befehlen wird das CL-Register ebenfalls als Zähler benutzt. Mit CH und CL lassen sich die beiden 8-Bit-Anteile des Registers CX ansprechen.

### Das Daten-Register (DX)

Bei Ein-/Ausgaben zu den Portadressen läßt sich dieses Register als Zeiger auf den jeweiligen Port nutzen. Die Adressierung über DX ist erforderlich, falls Portadressen oberhalb FFH angesprochen werden. Weiterhin dient das Register DX zur Aufnahme von Daten bei 16-Bit-Multiplikations- und Divisionsoperationen. Mit DH und DL lassen sich die 8-Bit-Register ansprechen.

## 2.1.2 Die Index- und Pointer-Register

Die nächste Gruppe bilden die Index- und Pointer-Register SI, DI, BP, SP und IP. Die Register lassen sich nur mit 16-Bit-Breite ansprechen. Die Funktion der einzelnen Register wird nachfolgend kurz beschrieben.

### Der Source Index (SI)

Bei der Anwendung von String-Befehlen muß die Adresse des zu bearbeitenden Textes angegeben werden. Für diesen Zweck ist das Register SI (Source Index) vorgesehen. Es dient als Zeiger für die Adressberechnung im Speicher. Diese Berechnung erfolgt dabei zusammen mit dem DS-Register.

### Der Destination Index (DI)

Auch dieses Register wird in der Regel als Zeiger zur Adressberechnung verwendet. Bei String-Kopierbefehlen steht hier dann zum Beispiel die Zieladresse, an der der

Text abzuspeichern ist. Die Segmentadresse wird bei String-Befehlen aus dem ES-Register gebildet. Bei anderen Befehlen kombiniert die CPU das DI-Register mit dem DS-Register.

### Das Base Pointer-Register (BP)

Hierbei handelt es sich ebenfalls um ein Register, welches zur Adressberechnung benutzt wird. Im Gegensatz zu den Zeigern BX, SI und DI wird die physikalische Adresse (Segment + Offset) aber zusammen mit dem Stacksegment SS gebildet. Dies bringt insbesondere bei der Parameterübergabe in Hochsprachen Vorteile, falls diese auf dem Stack abgelegt werden.

### Der Stackpointer (SP)

Dieses Register wird speziell für die Verwaltung des Stacks benutzt. Beim Stack handelt es sich um eine Speicherstruktur, in der sich Daten nur sequentiell speichern lassen. Lesezugriffe beziehen sich dabei immer auf den zuletzt gespeicherten Wert. Erst wenn dieser Wert vom Stack entfernt wurde, läßt sich auf den nächsten Wert zugreifen. Der Stack dient zur Aufnahme von Parametern und Programmrücksprung-adressen. Näheres wird im Rahmen der CALL-, PUSH-, POP- und RET-Befehle erläutert.

### Der Instruction Pointer (IP)

Dieses Register wird intern von der CPU verwaltet und steht für den Programmierer nicht zur Verfügung. Der Inhalt wird zusammen mit dem Codesegmentregister CS genutzt, um die nächste auszuführende Instruktion zu markieren. Der Wert wird durch Unterprogrammaufrufe, Sprünge und RET-Befehle beeinflußt. Näheres findet sich bei der Vorstellung der CALL- und JMP-Befehle.

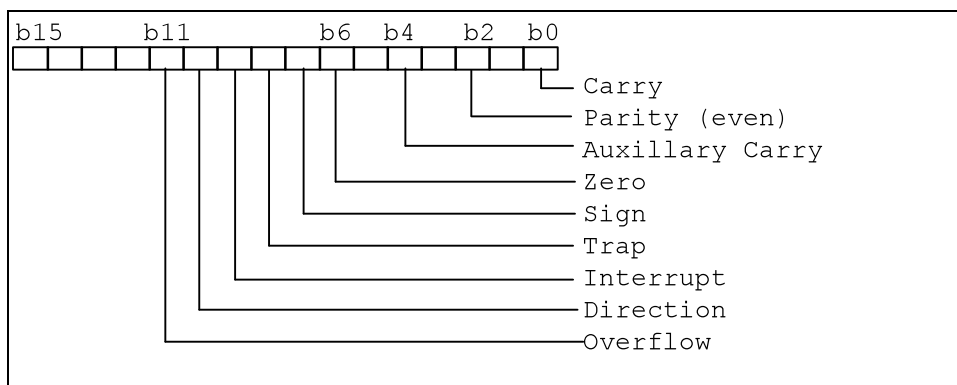


Bild 2.2: Die Kodierung der 8086-Flags

### 2.1.3 Die Flags

Der Prozessor besitzt ein eigenes 16 Bit breites Register, welches in einzelne Flagbits unterteilt wird. Mit diesen Flags zeigt die CPU das Ergebnis verschiedener Operationen an. In Bild 2.2 ist die Kodierung dieses Registers dokumentiert.

Nachfolgend wird die Bedeutung der einzelnen Bits des Flagregisters vorgestellt.

#### Das Carry-Flag (CF)

Das Carry-Flag wird durch Additionen und Subtraktionen bei 8- oder 16-Bit-Operationen gesetzt, falls ein Überlauf auftritt oder ein Bit geborgt werden muß. Weiterhin beeinflussen bestimmte Rotationsbefehle das Bit.

#### Das Auxillary-Carry-Flag

Ähnliches gilt für das Auxillary-Carry, welches bei arithmetischen Operationen gesetzt wird, falls ein Überlauf zwischen den unteren und den oberen vier Bits eines Bytes auftritt. Das gleiche gilt, falls ein Bit geborgt werden muß.

#### Das Overflow-Flag (OF)

Mit dem Overflow-Flag werden arithmetische Überläufe angezeigt. Dies tritt insbesondere auf, falls signifikante Bits verloren gehen, weil das Ergebnis einer Multiplikation nicht mehr in das Zielregister paßt. Weiterhin existiert für diesen Zweck ein Befehl (Interrupt on Overflow), der zur Auslösung einer Unterbrechung dienen kann.

#### Das Sign-Flag (SF)

Das Sign-Flag ist immer dann gesetzt, falls das oberste Bit des Ergebnisregisters 1 ist. Falls dieses Ergebnis als Integerzahl in der Zweierkomplementdarstellung gewertet wird, ist die Zahl negativ.

#### Das Parity-Flag (PF)

Bei der Übertragung von Daten ist die Erkennung von Fehlern wichtig. Oft erfolgt dies durch eine Paritätsprüfung. Hierbei wird festgestellt, ob die Zahl der binären Einsen in einem Byte gerade oder ungerade ist. Bei einem gesetzten Parity-Flag ist die Zahl der Einsbits gerade (even).

### Das Zero-Flag (ZF)

Mit diesem Bit wird angezeigt, ob das Ergebnis einer Operation den Wert Null annimmt. Dies kann zum Beispiel bei Subtraktionen der Fall sein. Weiterhin läßt sich eine Zahl mit der AND-Operation (z.B. AND AX,AX) auf Null prüfen. Ein gesetztes Flag bedeutet, daß Register AX enthält den Wert Null.

### Das Direction-Flag (DF)

Bei den String-Befehlen muß neben der Anfangsadresse und der Zahl der zu bearbeitenden Bytes auch die Bearbeitungsrichtung angegeben werden. Dies erfolgt durch das Direction-Bit, welches sich durch eigene Befehle beeinflussen läßt. Bei gesetztem Bit wird der Speicherbereich in absteigender Adreßfolge bearbeitet, während bei gelöschtem Bit die Adressen nach jeder Operation automatisch erhöht werden.

### Das Interrupt-Flag (IF)

Die CPU prüft nach jedem abgearbeiteten Befehl, ob eine externe Unterbrechungsanforderung am Interrupt-Eingang anliegt. Ist dies der Fall, wird das gerade abgearbeitete Programm unterbrochen und die durch einen besonderen Interrupt-Vektor spezifizierte Routine ausgeführt. Bei einem gelöschten Bit werden diese externen Unterbrechungsanforderungen ignoriert.

### Das Trap-Flag (TF)

Mit diesem Bit läßt sich die CPU in einen bestimmten Modus (single step mode) versetzen. Dies bedeutet, daß nach jedem ausgeführten Befehl ein INT 1 ausgeführt wird. Dieser Modus wird insbesondere bei Debuggern ausgenutzt, um Programme schrittweise abzuarbeiten.

Die restlichen Bits im Flagregister sind beim 8086-Prozessor unbelegt.

## 2.1.4 Die Segmentregister

Damit bleiben nur noch die vier Register CS, DS, SS und ES übrig. Was hat es nun mit diesen Registern für eine Bewandnis? Hier spielt wieder die Architektur der 80x86-Prozessorfamilie eine Rolle. Alle CPU's können *im Real Mode* einen Adreßbereich von 1 MByte ansprechen. Dies ist auch der von MS-DOS verwaltete Speicher. Andererseits besitzt die CPU intern nur 16 Bit breite Register: Damit lassen sich jedoch nur 64 KByte adressieren. Um nun den Bereich von 1 MByte zu erreichen, benutzten die Entwickler einen Trick. Für 1 MByte sind 20-Bit-Adressen notwendig. Diese werden von der CPU aus zwei 16-Bit-Werten gemäß Bild 2.3 berechnet.

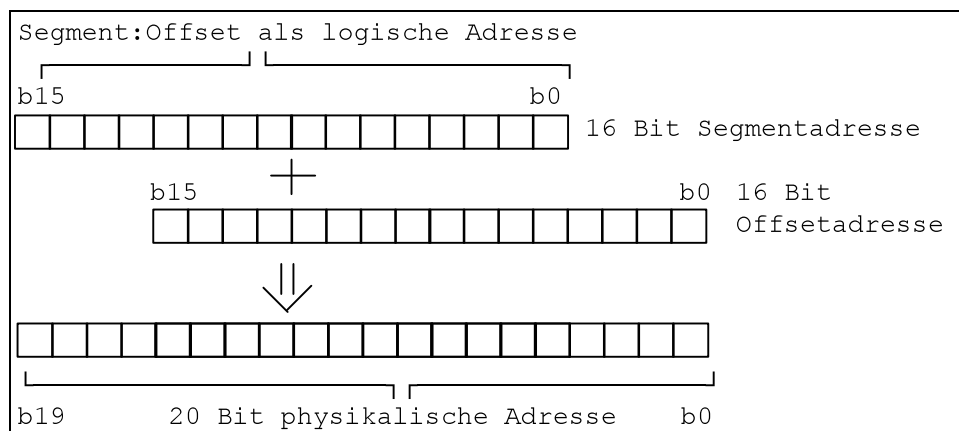


Bild 2.3: Adreßdarstellung in der Segment:Offset-Notation

Der Speicherbereich wird dabei einfach in Segmente von minimal 16 Byte (Paragraph) unterteilt. Dies ist zum Beispiel der Fall, wenn der Offsetanteil konstant auf 0000 gehalten wird, und die Segmentadresse mit der Schrittweite 1 erhöht wird. Der 1-MByte-Speicherraum zerfällt dann in genau 65536 Segmente.

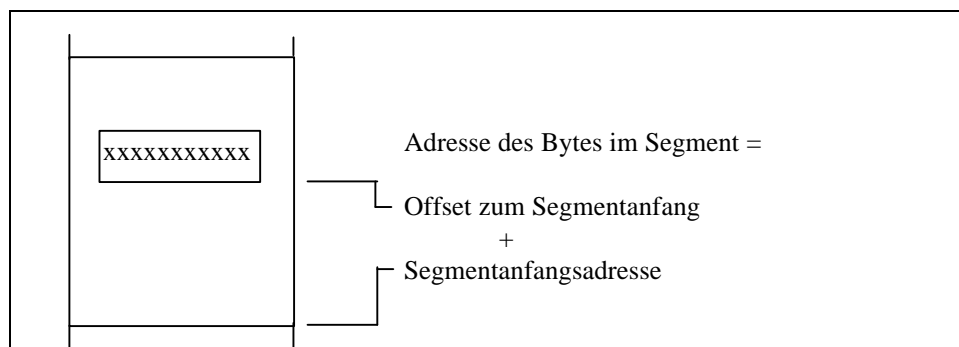


Bild 2.4: Angabe einer Speicheradresse mit Segment:Offset

Dieser Wert ist aber wieder mit einer 16-Bit-Zahl darstellbar. Mit Angabe der Segmentadresse wird also ein Speicherabschnitt (Segment) innerhalb des 1-MByte-Adressraumes angegeben. Die Adresse eines einzelnen Bytes innerhalb eines Segments läßt sich dann als Abstand (Offset) zum Segmentbeginn angeben (Bild 2.4).

Mit einer 16-Bit-Zahl lassen sich dann bis zu 65536 Byte adressieren. Damit liegt aber die Größe eines Segments zwischen 16 Byte (wenn der Offsetanteil auf 0000 gehalten wird) und 64 KByte (wenn der Offsetanteil zwischen 0000H und FFFFH variiert). Jede Adresse im physikalischen Speicher läßt sich damit durch Angabe der Segment- und der Offsetadresse eindeutig beschreiben. Die CPU besitzt einen internen

Mechanismus, um die 20 Bit physikalische Adresse automatisch aus der logischen Adresse in der Segment-Offset-Schreibweise zu berechnen.

In Anlehnung an Bild 2.3 werden alle Adressen im Assembler in dieser Segment-Offset-Notation beschrieben. Die physikalische Adresse F2007H kann dann durch die logische Adresse F200:0007 dargestellt werden. Der Wert F200 gibt die Segmentadresse an, während mit 0007 der Offset beschrieben wird. Alle Werte im nachfolgenden Text sind, sofern nicht anders spezifiziert, in der hexadezimalen Notation geschrieben. Die Umrechnung der logischen Adresse in die physikalische Adresse erfolgt durch eine einfache Addition. Dabei wird der Segmentwert mit 16 multipliziert, was im Hexadezimalsystem einer Verschiebung um eine Stelle nach links entspricht.

F200	Segment
<u>0007</u>	Offset
F2007	physikal. Adresse

Die Rückrechnung physikalischer Adressen in die Segment-Offset-Notation ist dagegen nicht mehr eindeutig. Die physikalische Adresse:

D4000

läßt sich mindestens durch die folgenden zwei logischen Adressen beschreiben:

D400:0000  
D000:0400

Die Umrechnung erfolgt am einfachsten dadurch, daß man die ersten 4 Ziffern (z.B. D400) zur Segmentadresse zusammenfaßt. Die verbleibende 5. Ziffer (z.B. 0) wird um 3 führende Nullen ergänzt und bildet dann die Offsetadresse. Alternativ läßt sich das Segment auch aus der ersten Ziffer (z.B. D), ergänzt um 3 angehängte Nullen, bestimmen. Dann ist der Offsetanteil durch die verbleibenden 4 Ziffern definiert. Beide Varianten wurden in obigem Beispiel benutzt. Die Probe auf eine korrekte Umrechnung der physikalischen in eine logische Adresse läßt sich leicht durchführen: Rechnen Sie einfach die logische Adresse in die physikalische Adresse um. Dann muß der ursprüngliche Wert wieder vorliegen. Andernfalls liegt ein Fehler vor. Tabelle 2.1 gibt einige Umrechnungsbeispiele an.

Segment:Offset	physikalische Adresse
F200:0000	F2007
F000:2007	F2007
F100:1007	F2007
D357:0017	D3587
0000:3587	03587

Tabelle 2.1: Umrechnung logische in physikalische Adressen.



Noch ein Hinweis: der Umgang mit dem Hexadezimalsystem ist bei der Assemblerprogrammierung absolut notwendig. Für die Einsteiger möchte ich nochmals eine kleine Umrechnungstabelle zwischen Hexadezimal- und Dezimalzahlen vorstellen.

Dezimal	Hexadezimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
14	E
15	F
16	10
128	80
255	FF

Tabelle 2.2: Umrechnung Hex-Dez

Doch nun zurück zu den vier Registern DS, CS, SS und ES, die zur Aufnahme der Segmentadressen dienen, weshalb sie auch als Segmentregister bezeichnet werden. Dabei besitzt jedes Register eine besondere Funktion.

### Das Codesegment-Register (CS)

Das Codesegment (CS) Register gibt das aktuelle Segment mit dem Programmcode an. Die Adresse der jeweils nächsten abzuarbeitenden Instruktion wird dann aus den Registern CS:IP gebildet.

### Das Datensegment-Register (DS)

Neben dem Code enthält ein Programm in der Regel auch Daten (Variable und Konstante). Werden diese nun mit im Codesegment abgelegt, ist der ganze Bereich auf 64 KByte begrenzt. Um flexibler zu sein, haben die Entwickler ein eigenes Segmentregister für die Daten vorgesehen. Alle Zugriffe auf Daten benutzen implizit das DS-Register zur Adressberechnung.

### Das Stacksegment-Register (SS)

Auch der Stack läßt sich in einen eigenen Bereich legen. Die Stackadresse wird von der CPU automatisch aus den Werten SS:SP berechnet.

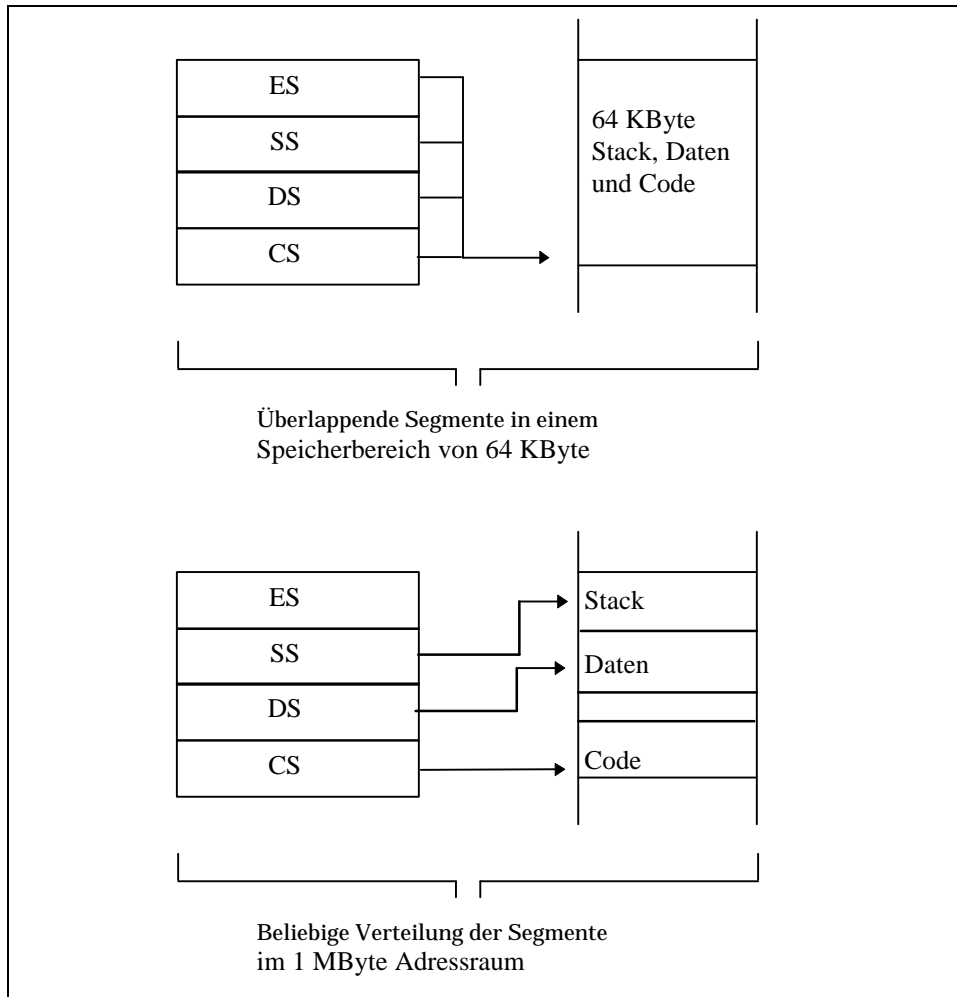


Bild 2.5: Verschiebung der Segmente im Adreßraum

### Das Extrasegment-Register (ES)

Dieses Register nimmt eine Sonderstellung ein. Normalerweise sind bereits alle Segmente für Code, Daten und Stack definiert. Bei Stringcopy-Befehlen kann es aber durchaus vorkommen, daß Daten über einen Segmentbereich hinaus verschoben werden müssen. Das DI-Register wird bei solchen Operationen automatisch mit dem ES-Register kombiniert. Die Verwendung der Segmentregister bringt natürlich den

Nachteil, daß alle Programm- oder Datenbereiche größer als 64 KByte in mehrere Segmente aufzuteilen sind. Bei Prozessoren mit einem 32-Bit-Adressregister lassen sich wesentlich größere lineare Adreßräume erzeugen. Aber die Segmentierung hat auch seine Vorteile. Wird bei der Programmierung darauf geachtet, daß sich alle Adressangaben relativ zu den Segmentanfangsadressen beziehen (relative Adressierung), ist die Software in jedem beliebigen Adreßbereich lauffähig (Bild 2.5).

Die Segmentregister müssen damit erst zur Laufzeit gesetzt werden. Bei der Speicherverwaltung durch ein Betriebssystem ist dies recht vorteilhaft. So legt MS-DOS die Segmentadressen erst zur Ladezeit eines Programmes fest. Bei COM-Dateien enthalten zum Beispiel alle Segmentregister den gleichen Startwert, womit Code, Daten und Stack in einem 64-KByte-Segment liegen. Diese Struktur wurde von CP/M übernommen, wo auch nur 64 KByte zur Verfügung standen.

Allerdings gibt es einige Einschränkungen bezüglich der Ladeadressen der Segmente. Die Entwickler der 8086 CPU haben im Speicher zwei Bereiche für andere Aufgaben reserviert (Bild 2.6).

Die obersten 16 Byte, von FFFF:0000 bis FFFF:000F, dienen zur Aufnahme des Urstartprogramms. Nach jedem Reset beginnt die CPU ab der Adresse FFFF:0000 mit der Abarbeitung der ersten Befehle. Bei PCs befindet sich an dieser Stelle dann das BIOS-ROM.

Der andere reservierte Bereich beginnt ab Adresse 0000:0000 und reicht bis 0000:03FF. In diesem 1 KByte großen Bereich verwaltet der Prozessor die insgesamt 256 Interrupt-Vektoren.

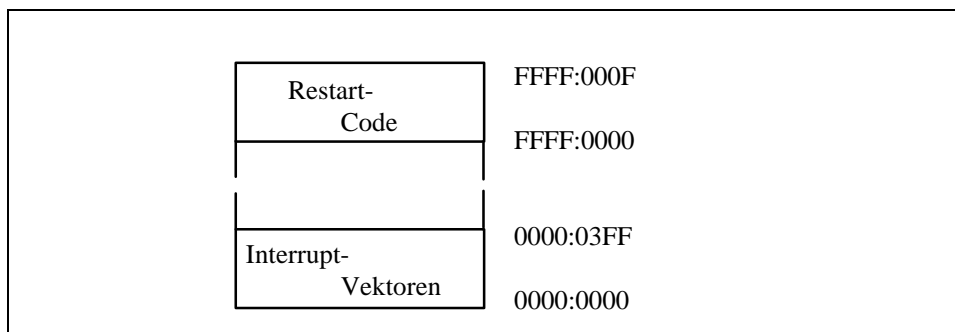


Bild 2.6: Reservierte Adressbereiche

Hierbei handelt es sich um eine Tabelle mit 255 4-Byte-Adressen der Routinen, die bei einer Unterbrechung zu aktivieren sind. Im Rahmen der Vorstellung der INT-Befehle wird dieser Aspekt noch etwas detaillierter behandelt.

Zum Abschluß noch ein Hinweis zur Abspeicherung der Daten im Speicher. Von der Adressierung her wird der Speicherbereich in Bytes unterteilt. Soll nun aber eine 16-Bit-Zahl gespeichert werden, ist diese in zwei aufeinanderfolgenden (Bytes) Speicher-

zellen unterzubringen. Dabei gilt, daß das untere Byte der Zahl in der unteren Adresse abgelegt wird. Bei der Ausgabe in Bytes erscheint das unterste Byte allerdings zuerst (Bild 2.7).

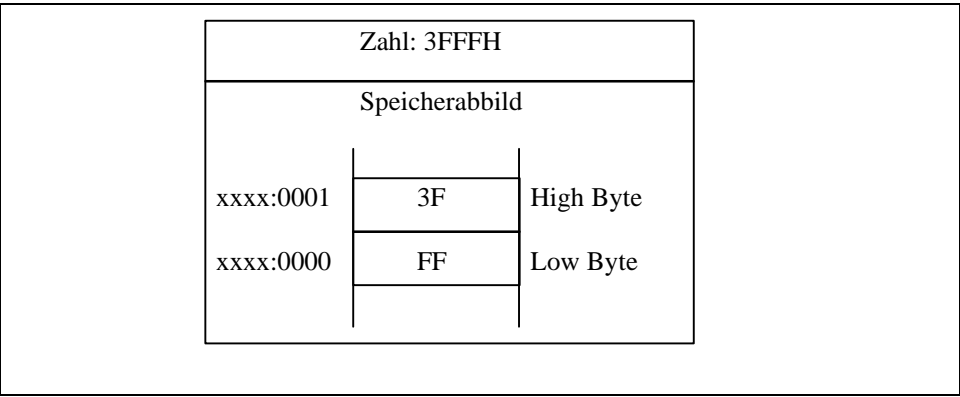


Bild 2.7: Speicherabbildung von 16-Bit-Zahlen

Das Wort 3FFF wird demnach als Bytefolge FF 3F auf dem Bildschirm ausgegeben. Dies sollten sich insbesondere die Einsteiger gut merken, da sonst einige Probleme auftreten können.

2.2 Die Befehle der 8086-CPU

Nach diesen Vorbemerkungen können wir uns den eigentlichen Befehlen des Prozessors zuwenden. Der Befehlssatz umfaßt eine Reihe von Anweisungen zur Arithmetik, zur Behandlung der Register, etc. Tabelle 2.3 gibt die einzelnen Befehlsgruppen wieder.

Gruppe	Befehle
Daten Transfer Befehle:	MOV, PUSH, POP, IN, OUT, etc.
Arithmetik Befehle:	ADD, SUB, MUL, IMUL, DIV, IDIV, etc.
Bit Manipulations Befehle:	NOT, AND, OR, XOR, SHR, ROL, etc.
Gruppe	Befehle
Programm Transfer Befehle:	CALL, RET, JMP, LOOP, INT, IRET, etc.
Prozessor Kontroll Befehle:	NOP, STC, CLI, HLT, WAIT, etc.

Tabelle 2.3: Gruppierung der 8086-Befehle

In den nachfolgenden Abschnitten werden diese Befehle detailliert vorgestellt.

## 2.3 Die 8086-Befehle zum Datentransfer

Die Gruppe umfaßt die Befehle zum allgemeinen Datentransfer (MOV, PUSH, POP, etc.). Als erstes wird der MOV-Befehl im vorliegenden Abschnitt besprochen.

### 2.3.1 Der MOV-Befehl

Einer der Befehle zum Transfer von Daten ist der MOV-Befehl. Er dient zum Kopieren von 8- und 16-Bit-Daten zwischen den Registern und zwischen Registern und Speicher. Dabei gilt folgende Syntax:

MOV Ziel,Quelle

Mit den drei Buchstaben *MOV* wird der Befehl mnemotechnisch dargestellt, während die Parameter *Ziel* und *Quelle* als Operanden dienen. *Ziel* gibt dabei an, wohin der Wert zu speichern ist. Mit dem Operanden *Quelle* wird spezifiziert, von wo der Wert zu lesen ist. Befehl und Operanden sollten mindestens durch ein Leerzeichen getrennt werden, um die Lesbarkeit zu erhöhen. Die Parameter selbst sind durch ein Komma zu separieren.

Die Anordnung der Operanden entspricht übrigens der gängigen Schreibweise in vielen Programmiersprachen, wo bei einer Zuweisung das Ziel auch auf der linken Seite steht:

Ziel := Quelle;

Alle INTEL-Prozessoren halten sich an diese Notation. Es sei aber angemerkt, daß es durchaus Prozessoren gibt, bei denen der erste Operand auf den zweiten Operand kopiert wird. Es gibt nun natürlich eine Menge von Kombinationen (28 Befehle) zur Spezifikation der beiden Operanden. Diese werden nun sukzessive vorgestellt.

#### MOV-Befehle zwischen Registern

Die einfachste Form des Befehls benutzt nur Register als Operanden. Dabei gilt folgende Form:

MOV Reg1,Reg2

Mit *Reg1* wird das Zielregister und mit *Reg2* das Quellregister spezifiziert. Der Befehl kopiert dann die Daten von *Register 2* nach *Register 1*. Dabei lassen sich sowohl 8-Bit- als auch 16-Bit-Register angeben (Tabelle 2.4).

MOV AX,BX	MOV AH,AL
MOV DS,AX	MOV BH,AL
MOV AX,CS	MOV DL,DH
MOV BP,DX	MOV AL,DL

Tabelle 2.4: Register-Register-MOV-Befehle

Der Assembler erkennt bei AX, BX, CX und DX die Registerbreite an Hand des letzten Buchstabens. Ist dieser ein X, wird automatisch ein 16-Bit- Universalregister (z.B. AX) benutzt. Wird dagegen ein H oder L gefunden, bezieht sich der Befehl auf das High- oder Low-Byte des jeweiligen Registers (z.B. AH oder AL).

Bei den MOV-Befehlen wird der Inhalt des Quelle zum Ziel kopiert, der Wert der Quelle bleibt dabei erhalten. Dies wird an folgenden Bildern deutlich. Im ersten Schritt (Bild 2.8) seien die Register mit folgenden Daten vorbelegt.

AX	003F
BX	3FFF
CX	1234

Bild 2.8: Registerinhalt vor Ausführung des Befehls

AX	3F3F
BX	1234
CX	1234

Bild 2.9: Registerinhalt nach der Ausführung der Befehle

Nun führt die CPU folgende Befehle aus:

```
MOV AH,AL
MOV BX,CX
```

Danach ergibt sich in den Registern die Belegung gemäß Bild 2.9.

Versuchen Sie ruhig einmal obiges Beispiel mit dem DOS-Debugger nachzuvollziehen. Geben Sie dazu in DEBUG die nachfolgenden Anweisungen ein:

```
A 100
MOV AH,AL
MOV BX,CX
INT 3
```

```
R
G = 100 103
R
```

Der INT 3-Befehl sorgt dafür, daß der Debugger nicht über das Ende des Programmes hinausläuft. Zwischen *INT 3* und *R* muß eine Leerzeile eingefügt werden. Mit dem G-Kommando werden die Befehle ausgeführt. Nähere Hinweise zum Umgang mit DEBUG.COM finden sich im Anhang.

**Warnung:** Im MOV-Befehl lassen sich als Operanden alle Universalregister, die Segmentregister und der Stackpointer angeben. Allerdings gibt es bei der Anwendung des Befehls auch einige Einschränkungen. So sollte die folgende Anweisung nach Möglichkeit vermieden werden:

```
MOV CS,AX
```

Mit dem Befehl wird das Codesegmentregister mit dem Inhalt von AX überschrieben. Dies führt dazu, daß der Prozessor bei der folgende Anweisung auf ein neues Codesegment zugreift. Die Auswirkungen gleichen einem Sprungbefehl an eine andere Programmstelle. Da aber der Instruction-Pointer (IP) nicht mit verändert wurde, sind die Ergebnisse meist undefiniert. Dem Befehl ist dieser Seiteneffekt nicht anzusehen. Vielleicht versuchen Sie diesen Effekt einmal im DOS-Debugger mit folgendem Programm nachzuvollziehen:

```
A 100
MOV AX,0000
MOV CS,AX
INT 3
```

```
G = 100 105
```

Nach der Ausführung der Sequenz zeigt das CS-Register auf den unteren 64-KByte-Bereich des Speichers. Der nächste Befehl wird dann aus diesem Codesegment gelesen. Die INT 3-Anweisung wird also nie mehr erreicht. Das Ergebnis der Zuwei-

sung ist in der Regel ein Systemabsturz. Um die Programmausführung in einem anderen Segment zu erreichen, gibt es leistungsfähigere Anweisungen (CALL, JMP, etc.), die in späteren Abschnitten noch detailliert behandelt werden.

Weiterhin sind folgende Befehlskombinationen unzulässig:

- ◆ Verwendung des Registers IP als Operand (z.B. MOV IP,AX)
- ◆ Gemischte Verwendung von 8- und 16-Bit-Registern (z.B. MOV AX,BL)

Der Assembler wird diese Befehle mit einer Fehlermeldung zurückweisen. Versuchen Sie ruhig einmal die zulässigen Kombinationen mittels DEBUG herauszufinden.

### Der Immediate-MOV-Befehl

Auf die Dauer wird es etwas langweilig, nur Daten zwischen den Registern hin und her zu kopieren. Um die Register mit gezielten Werten zu besetzen, benötigt die CPU die Möglichkeit, diese Daten direkt aus dem Speicher zu lesen. Hierfür findet der Immediate-MOV-Befehl Verwendung. Dabei gilt die gleiche Syntax wie bei den bereits besprochenen Befehlen:

MOV Ziel,Konstante

Als Quelle wird dann eine Konstante direkt (immediate) aus dem Speicher gelesen und in das Ziel kopiert. Als Ziel läßt sich ein Register oder eine Speicherstelle angeben. Die Datenbreite der Konstanten wird durch das Ziel bestimmt. Bei 16-Bit-Registern als Ziel wird immer eine 16-Bit-Konstante gelesen. Nachfolgend werden einige gültige Befehle angegeben.

```
MOV AX,0000
MOV AH,3F
MOV BYTE [3000],3F
MOV WORD [4000],1234
MOV BP,1400
```

Bei der Verwendung von Registern als Ziel ist der Wertebereich der Konstanten festgelegt. So liest der erste Befehl (MOV AX,0000) den Wert 0000H in das 16-Bit-Register AX ein. Im zweiten Befehl (MOV AH,3F) darf nur eine Bytekonstante eingesetzt werden, da ja das Register AH genau 8 Bit breit ist. Die Anweisung:

MOV AH,3FFF

führt zu einer Fehlermeldung des Assemblers, da der Wert nicht in das Register paßt. Erlaubt sind allerdings führende Nullen (MOV AH,0FF). Der Wert der Konstanten darf aber den vorgegebenen Bereich nicht überschreiten. Werden bei einer 16-Bit-Konstanten weniger als vier Ziffern eingegeben (MOV AX,01), ersetzt der Assembler



die führenden Stellen durch Nullen. Das Beispiel aus Bild 2.10 gibt die Registerbelegung vor einem Immediate-MOV-Befehl an.

AX	3F3F
BX	1234
CX	1234

*Bild 2.10: Registerinhalt vor den Immediate-MOV-Befehlen*

Das Register AX soll gelöscht und BH mit FFH belegt werden. Dies ist mit folgender Sequenz möglich.

```
MOV AX,0000
MOV BH,0FF
```

Nach Ausführung der Befehle ergeben sich die Registerinhalte gemäß Bild 2.11.

AX	0000
BX	FF34
CX	1234

*Bild 2.11: Registerinhalt nach den Immediate-MOV-Befehlen*

Die Funktionen der Befehle die sich auf Register als Ziel beziehen sind wohl intuitiv klar. Nun tauchen in obigem Beispiel aber auch Zuweisungen von Konstanten an Speicherstellen auf. Der Befehl:

```
MOV BYTE [3000],3F
```

überschreibt nicht den Wert 3000 mit 3F, sondern speichert die Konstante 3FH im Speicher an der Adresse DS:[3000] ab. Dies wird dadurch signalisiert, daß die Adresse in eckige Klammern [] gesetzt ist. Mit der Anweisung:

```
MOV [3000],03F
```

kann der Assembler aber noch nichts anfangen, da nicht feststeht, ob ein Byte oder ein Wort zu kopieren ist. Deshalb erwarten viele Assembler, daß der Programmierer

explizit die Breite (BYTE oder WORD) angibt. Dies kann durch folgende Anweisungen geschehen:

```
MOV BYTE [3000],03F
MOV BYTE PTR [3000],03F
MOV WORD [4000],7FFF
MOV WORD PTR [4000],7FFF
```

Obige Anweisungen veranlassen die Zuweisung der Bytekonstanten 3F auf das Byte ab Adresse DS:3000 und der Word-Konstanten 7FFFH auf die Adresse DS:4000. Die Anweisung PTR kann bei dem Programm DEBUG entfallen, da nur die Schlüsselworte BYTE oder WORD ausgewertet werden. Nur bei Zuweisungen an Variable steht der Typ fest. Dieser Modus wird zwar durch MASM, nicht aber durch DEBUG unterstützt. Der Wert in Klammern gibt den Offset innerhalb eines Segments an. Dieser Offset bezieht sich dabei standardmäßig auf das Datensegment (DS).

Der Immediate-MOV-Befehl kann auf die Register:

AX,BX,CX,DX,BP,SP,SI,DI

angewandt werden. Nicht möglich ist es, die immediate Konstante:

```
MOV 3FFF,AX
```

als Ziel anzugeben. Dies ergibt auch keinen Sinn, da dann ja eine Konstante durch den Inhalt eines Registers überschrieben würde. Weiterhin sind direkte Zuweisungen von Konstanten an Segmentregister wie:

```
MOV DS,3500
```

unzulässig. Um ein Segmentregister mit einer Konstanten zu laden, ist der Umweg über ein Universalregister oder die indirekte Adressierung zu wählen (s. folgende Beispiele).

### Programmbeispiel

Damit kommen wir zu unserem ersten kleinen Programmbeispiel. Es soll versucht werden, direkt in den Bildschirmspeicher zu schreiben. Der Bildschirmspeicher beginnt bei Monochromkarten auf der Segmentadresse B000H. Falls der PC einen CGA-Adapter besitzt, liegt die Segmentadresse des Bildschirmspeichers bei B800H. Jedes angezeigte Zeichen belegt im Textmodus zwei Byte im Bildschirmspeicher. Im ersten Byte steht der ASCII-Code des Zeichens (z.B. 41H für den Buchstaben 'A'). Das Folgebyte enthält das Attribut für die Darstellung (fett, invers, blinkend, etc.). Die Kodierung der Attribute ist in /2/ angegeben. Der Wert 07H steht für eine normale Darstellung, während mit 7FH die Anzeige invers erfolgt. Erstellen Sie mit einem Texteditor eine Quelldatei mit dem Namen:

## SCREEN.ASM

und übersetzen diese Quelldatei mit DEBUG über folgende Anweisung:

DEBUG < SCREEN.ASM > SCREEN.LST

Die Quelldatei muß genau nach den Vorgaben des folgenden Listings aufgebaut sein. Leerzeilen sind peinlich genau an den markierten Stellen einzubringen. DEBUG wird eine Listdatei in SCREEN.LST anlegen. Sofern diese Datei keine Fehlermeldungen enthält existiert anschließend auf dem Standardlaufwerk ein ablauffähiges COM-Programm mit dem Namen SCREEN.COM. Weitere Hinweise über den Umgang mit DEBUG finden sich in einem der folgenden Kapitel.

```
A 100
;=====
; File: SCREEN.ASM (c) G. Born
; Aufgabe: Ausgabe des Buchstabens A auf
; dem Bildschirm. Es wird direkt in den
; Speicher ab der Segmentadresse geschrie-
; ben. Die Adresse liegt bei B000H (mono)
; oder B800 (color).
;=====
MOV AX,B800      ; Seg. Adr.
MOV DS,AX        ; Screen setzen
MOV BYTE [0500],41 ; 'A'
MOV BYTE [0501],7F ; Attribut
INT 3 ; hier muss eine Leerzeile folgen

R CX
200
N SCREEN.COM
W
Q
```

*Listing 2.1: Screenausgabe*

Gegebenenfalls lassen sich die MOV-Befehle auch direkt in DEBUG eingeben. Hierzu ist DEBUG aufzurufen und der A-Befehl ist zu aktivieren. Dann können einzelne Assembleranweisungen, allerdings ohne den Text hinter den Semikolons, eingegeben werden.

Die Texte hinter den Semikolons sind Kommentare, sie werden von DEBUG überlesen und müssen nicht mit angegeben werden.

Falls Sie das Programm mit DEBUG < SCREEN.... übersetzt haben, laden Sie die COM-Datei mit:

DEBUG SCREEN.COM

und starten das Programm mit der Anweisung:

G = 100 10F

Mit dem ersten Assemblerbefehl (MOV AX, B800) wird die Segmentadresse des Bildschirmspeichers definiert. Diese liegt bei B000 oder B800 und ist in das DS-Register zu kopieren. Anschließend lassen sich die Konstanten 41H und 7FH auf die Adressen DS:0500H und DS:0501H schreiben, denn der MOV-Befehl benutzt beim Zugriff auf den Speicher automatisch das DS-Register als Segment. Die Anweisung MOV BYTE [0500],41 weist also einer Zelle im Bildschirmspeicher die Konstante 41H zu. Der Wert in Klammern ([0500]) gibt dabei den Offset innerhalb des Segments an.

Obiges Beispiel gibt den Buchstaben 'A' (Code 41H) auf dem Bildschirm aus. Durch die Wahl der Adressen 500 und 501 wird das Zeichen ab der 9. Zeile ausgegeben, so daß auch ein Bildscroll die Ausgabe nicht sofort überschreibt. Da das Attribut gleichzeitig auf den Wert 7FH gesetzt wird, sollte die Anzeige invers erscheinen. Falls das Beispiel bei Ihrem PC nicht funktioniert, prüfen Sie bitte, ob die Segmentadresse korrekt gesetzt wurde.

Noch ein Trick am Rande: falls Sie das DOS-Programm DEBUG benutzen, geben Sie als letzte Anweisung (sofern nichts anderes spezifiziert wird) immer die Instruktion INT 3 ein. Diese dient DEBUG als Unterbrechungspunkt. Dadurch reicht es, den Programmablauf mit:

G = 100

zu starten. Beim Erreichen der INT 3-Anweisung terminiert das Programm und der Debugger meldet sich mit dem Prompt - zurück. Dadurch wird auch bei nicht korrekt angegebener Endadresse das Programm beendet. Beachten Sie aber, daß das obige Testprogramm nicht alleine ohne DEBUG läuft. In späteren Programmbeispielen lernen Sie noch Techniken kennen um ein Assemblerprogramm korrekt zu beenden, so daß die Kontrolle an DOS zurückgegeben wird

### Indirekte Adressierung beim MOV-Befehl

Bisher wurden beim MOV-Befehl die Register und Konstanten direkt angegeben. Oft möchte der Programmierer jedoch ein Ergebnis aus einem Register auf eine Speicherstelle zurückspeichern. Hier besteht die Notwendigkeit innerhalb des MOV-Befehls eine Speicherstelle angeben zu müssen. Eine Variation haben wir bereits bei der Diskussion des Immediate-MOV-Befehls kennengelernt. Der Befehl:

MOV BYTE [3000],03F

schreibt eine Konstante an die Adresse DS:[3000]. Die Zieladresse wird dabei direkt angegeben. Dies bringt aber den Nachteil, daß bei Zugriffen auf den Speicher die Adressen bereits bei der Programmierung bekannt sein müssen. Dies ist natürlich recht unflexibel, falls Adressen während der Programmlaufzeit zu berechnen sind. Denken

Sie an eine Tabelle, wo ein Zugriff auf verschiedene Elemente über einen Index erfolgt. Hier bietet der 8086-Prozessor die Möglichkeit der indirekten Adressierung über Register. Dabei gilt die bereits vorgestellte Aufrufsyntax:

MOV Ziel, Quelle

Wobei als Ziel ein Register oder ein Zeiger (Register und Konstante) angegeben werden darf. Als Quelle kommen Register, Konstante und Zeiger in Betracht. Mit Hilfe eines Zeigers lassen sich dann Speicherstellen adressieren, deren Inhalt (indirekt über den Zeiger) angesprochen wird. Nachfolgend sind einige gültige Befehle aufgeführt.

```
MOV AX,[3000]
MOV [3000],BX
MOV AX,[SI]
MOV DX,[3000+BX]
```

Die ersten beiden Anweisungen enthalten noch absolute Adressangaben und wurden bereits in ähnlicher Form beim Immediate-MOV-Befehl vorgestellt. Bei den folgenden zwei Befehlen deuten sich aber bereits die Möglichkeiten der indirekten Adressierung an. Die Adresse der Speicherstelle wird indirekt über ein Register (SI) oder über einen Ausdruck (3000+BX) angegeben. Dies bedeutet, der Prozessor muß sich die Adresse erst aus dem Ausdruck in den eckigen Klammern [] berechnen. Bei der letzten Anweisung wird demnach der Inhalt des Registers BX zur Konstante 3000 addiert. Das Ergebnis bildet dann die Speicherstelle, deren Wert in das Register DX gelesen wird. Da DX ein 16-Bit-Register darstellt, wird ein Word (2 Byte) gelesen (Bild 2.12).

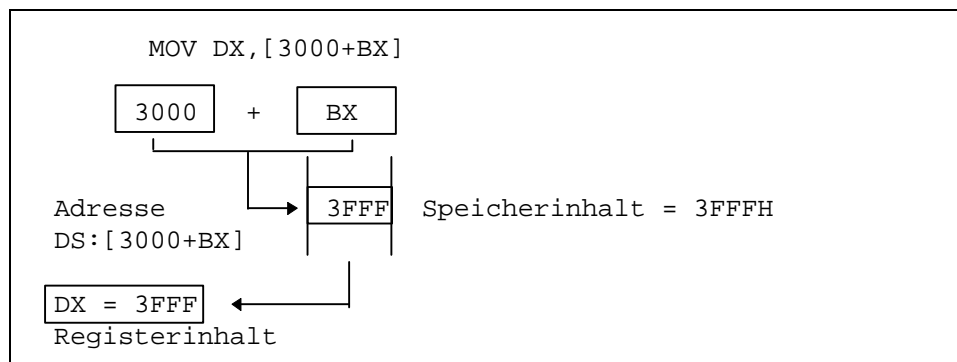


Bild 2.12: Indirekte Adressierung beim MOV-Befehl

In Bild 2.12 wird ein Zeiger aus der Konstanten (3000H) und dem Inhalt des BX-Registers gebildet. Das Ergebnis wird mit DS als Zeiger in den 1-MByte-Speicherbereich des Prozessors genutzt. An dieser Adresse soll nun der Wert 3FFFH stehen. Der Prozessor liest die beiden Bytes und kopiert sie in das Register DX. Nach Aus-

führung der Anweisung enthält das Register DX dann den Inhalt des Wortes an der Adresse DS:[3000+BX]. Tabelle 2.5 gibt die Möglichkeiten zur Berechnung der indirekten Adressen an.

BX + SI + DISP
BX + DI + DISP
BP + SI + DISP
BP + DI + DISP
SI + DISP
DI + DISP
BP + DISP
BX + DISP
DI
SI
BX
BP
DISP

Tabelle 2.5: Indirekte Adressierung über Register

Mit DISP wird dabei eine Konstante (Displacement) angegeben. Die Adresse errechnet sich dabei immer aus der Summe der Registerwerte und der eventuell vorhandenen Konstante. Als Zieloperand dürfen sowohl die Universalregister AX, BX, CX und DX, als auch die Segmentregister angegeben werden. Weiterhin ist auch die Ausgabe auf Speicherstellen über indirekte Adressierung möglich (z.B. MOV [SI]+10,AX). Das gleiche gilt für den Quelloperanden. Damit sind zum Beispiel folgende Befehle zulässig:

```
MOV DS,[BP]
MOV [BP + SI + 10], SS
MOV AH,[DI+3]
```

An dieser Stelle möchte ich noch einige Bemerkungen zur Syntax der indirekten Befehle geben. Der Assembler muß erkennen, daß es sich um eine Adreßangabe handelt. Als Zeiger können nun Konstanten und verschiedene Register auftreten. Leider unterscheidet sich hier die Syntax zur Eingabe solcher Befehle von Assembler zu Assembler. Das Programm DEBUG reagiert hier allerdings recht flexibel:

```
MOV AX,100[BP+SI]
MOV AX,[BP+SI]100
MOV AX,[BP][SI]100
MOV AX,[100][BP][SI]
MOV AX,[100+BP+SI]
MOV AX,[100][BP+SI]
```

Die aufgeführten Anweisungen werden alle als gültig akzeptiert. Dadurch wird das AX-Register mit dem Wert der durch den Zeiger BP+SI+100 adressierten Speicher-

zelle geladen. Es werden zwei Byte gelesen, da AX ein 16-Bit-Register ist. Experimentieren Sie ruhig etwas mit DEBUG um die erlaubten Kombinationen herauszufinden.

**Achtung:** Bei der Verwendung der verschiedenen Register zur indirekten Adressierung ist allerdings noch eine Besonderheit zu beachten. Im allgemeinen beziehen sich alle Zugriffe auf die Daten im Datensegment, benutzen also das DS-Register. Wird das Register BP innerhalb des Adreßausdruckes benutzt, erfolgt der Zugriff auf Daten des Stacksegments. Es wird also das SS-Register zur Ermittlung des Segments benutzt. Bei Verwendung der indirekten Adressierung gilt daher die Zuordnung:

BP -> SS-Register

BX -> DS-Register

Die Anweisung:

MOV AX,[100 + BP]

liest die Daten von der Adresse SS:[100+BP] in das Register AX ein. Tabelle 2.6 enthält eine Zusammenstellung der jeweiligen Befehle und der zugehörigen Segmentregister.

Index-Register	Segment
BX + SI + DISP	DS
BX + DI + DISP	DS
BP + SI + DISP	SS
BP + DI + DISP	SS
SI + DISP	DS
DI + DISP	DS
BP + DISP	SS
BX + DISP	DS
DI	DS
SI	DS
BX	DS
BP	SS
DISP	DS

Tabelle 2.6: Segmentregister bei der indirekten Adressierung (Fortsetzung)

DISP steht hier für eine Konstante. Vielleicht stellen Sie sich nun ganz frustriert die Frage, wozu diese komplizierte Adressierung gebraucht wird? Die Entwickler haben mit der indirekten Adressierung eine elegante Möglichkeit zur Bearbeitung von Datenstrukturen geschaffen. Hochspracheprogrammierer werden sicherlich die folgende (PASCAL) Datenstruktur kennen:

```

Type Adr = Record
  Name : String[20];
  PLZ : Word;
  Ort : String[20];
  Strasse : String[20];
  Nr : Word;
end;

```

```

Var
  Adresse : Array [0..5] of Adr;

```

Der Übersetzer legt diese Struktur an einer Adresse als zusammenhängendes Gebilde im Datenbereich ab. Um nun die einzelnen Elemente ansprechen zu können, muß die Adresse der jeweiligen Teilvariablen (z.B. Adresse[3].PLZ) berechnet werden. Hier zeigen sich nun die Stärken der indirekten Adressierung. Ein Register übernimmt die Basisadresse der Struktur, d.h. das Register bestimmt den Offset vom Segmentbeginn des Datenbereiches auf das erste Byte des Feldes Adresse[0].Name. Nun sind aber die einzelnen Feldelemente (Adresse[i].xx) anzusprechen. Es wird also ein zweiter Zeiger benötigt, der vom Beginn der Variablen Adresse[0].Name den Offset zum jeweiligen Feldelement Adresse[i].Name angibt. Dies erfolgt mit einem zweiten Register. Eine Konstante gibt dann den Offset vom Beginn der ersten Teilvariable Adresse[i].Name zum jeweiligen Element der Struktur (z.B. Adresse[i].Ort) an. Damit läßt sich zum Beispiel ein Zugriff auf einzelne Elemente mit folgender Konstruktion erreichen:

```

MOV BP, Adresse      ; Basisadresse
MOV SI,0              ; auf Adresse[0]
.
MOV AX,[BP+SI+14] ; get PLZ
MOV BX,[BP+SI+3E] ; get Nr.
.

```

In das Basisregister BP wird die Anfangsadresse der Variablen Adresse[0] geladen. Damit läßt sich aber nur auf das erste Byte der Struktur Adresse[0].Name zugreifen. Anschließend wird das Register SI als Index für die einzelnen Feldelemente (Adresse[i]) verwendet. Mit dem Wert SI=0 wird immer das Element Adresse[0].Name erreicht. Mit SI = 3EH erreicht man genau Adresse[1].Name, u.s.w. Soll nun ein Wert aus der Datenstruktur gelesen werden, kann die Adresse durch einen konstanten Offset vom Beginn des Elementes (Adresse[i].Name) bis zum jeweiligen Eintrag (z.B. Adresse[0].Ort) definiert werden. Alle Offsetwerte ergeben sich direkt aus der Definition der Datenstruktur. Damit lassen sich einzelne Felder durch einfache Veränderung des Registers SI bearbeiten (Bild 2.13).





MOV - Operanden	Beispiel
Register, Register	MOV AX, DX
Register, Speicher	MOV AX, [03FF]
Speicher, Register	MOV [BP+SI], DX
Speicher, Akkumulator	MOV 7FF[SI], AX
Akkumulator, Speicher	MOV AX, [BX]300
Register, immediate	MOV AL, 03F
Speicher, immediate	MOV [30+BX+SI], 30
Seg. Reg., Reg. 16	MOV DS, DX
Seg. Reg., Speicher 16	MOV ES, [3000]
Register 16, Seg. Reg.	MOV BX, SS
Speicher 16, Seg. Reg.	MOV [BX], CS

Tabelle 2.7: Adressierungsarten des MOV-Befehls (Ende)

### Die Adressierungsarten für Speicherzugriffe

Für die Adressierungsarten des MOV-Befehls werden in der Literatur verschiedene Fachbegriffe benutzt, die ich nachfolgend kurz zusammenfassen möchte:

#### Direkte Adressierung

Dies ist die einfachste Adressierungsform um auf Daten aus dem Speicher zuzugreifen. Dabei wird eine Konstante in den Zieloperanden (z.B. MOV AL,03) geladen. Die Konstante (z.B. 03H) steht dabei im Codesegment zwischen den Programmanweisungen.

#### Register-indirekte Adressierung

Bei dieser Adressierungsart wird eines der Register BX, BP, SI oder DI als Zeiger auf die Speicherzelle benutzt (z.B. MOV AX,[BX+SI]). Der Inhalt der Indexregister bestimmt zusammen mit dem Segmentregister die physikalische Speicheradresse. Normalerweise wird das Datensegment zum Zugriff benutzt. Taucht aber BP als Indexregister auf, erfolgt der Zugriff auf den Speicher im Stacksegment.

### Basis Adressierung

Bei der Basis Adressierung handelt es sich um eine Variante der indirekten Adressierung über Register. Als Register dürfen aber nur BX und BP (z.B.: MOV AX,[BX]) verwendet werden. Lediglich Konstanten sind als Zusatz erlaubt.

### Index Adressierung

Auch diese Adressierungsart bildet eine Variante der indirekten Adressierung. Im Adreßausdruck sind allerdings nur die Indexregister SI und DI erlaubt. Damit sind Anweisungen wie:

```
MOV AX,[SI]
MOV CX,[DI+10]
MOV BX,[SI]
MOV DX,[SI+3]
```

möglich. Neben dem Indexregister darf lediglich eine Konstante als Displacement angegeben werden.

### Basis Index Adressierung

In dieser Adressierungsart dürfen Basis- und Indexregister, sowie Konstanten, verwendet werden (z.B. MOV AX,[BX+SI+10]). Der Prozessor bestimmt den Adressausdruck durch Addition der Einzelwerte und greift dann auf den Speicher zu.

### Der Segment-Override-Befehl

Der 8086-Befehlssatz benutzt für den Zugriff auf Daten jeweils ein Segmentregister um die zugehörige Segmentadresse festzulegen. Je nach Befehl gelangt dabei das CS-, DS- und SS-Register zum Einsatz. Konstante werden grundsätzlich aus dem Code-segment (CS) gelesen. Beim MOV-Befehl erfolgt der Zugriff auf das Datensegment (DS), sofern das Register BP nicht verwendet wird. Mit BP als Zeiger wird auf das Stacksegment (SS) zugegriffen. Die folgenden drei Befehle verdeutlichen diesen Sachverhalt nochmals:

```
MOV AX,3FFF      ; 3FFFH steht im Codesegment
MOV AX,[BX]      ; Zugriff über DS:[BX]
MOV AX,[BP]      ; Zugriff über SS:[BP]
```

Häufig möchte der Programmierer jedoch den Zugriff auf die Daten explizit über ein bestimmtes Segmentregister vornehmen und die Standardzuweisung außer Kraft setzen. Hier bietet der 8086-Befehlssatz die Möglichkeit das Segmentregister explizit vor dem Befehl anzugeben.

```
MOV AX,[BX+10]    ; Zugriff über das DS-Segment
ES:
MOV AX,[BX+10]    ; Zugriff über das ES-Segment
CS:
MOV DX,[BX+SI]    ; Zugriff über das CS-Segment
```

An den Segmentnamen ist ein Doppelpunkt anzufügen. Während die erste Anweisung noch die Standardsegmentierung benutzt, wird diese bei den zwei folgenden Befehlen außer Kraft gesetzt. Die Zugriffe erfolgen über ES und über CS.

Diese Technik wird als *Segment Override* bezeichnet. Vor den eigentlichen Befehl wird die Segment-Override-Anweisung (DS:, ES:, CS:, SS:) gestellt. Der Segment-Override gilt jeweils nur für den direkt folgenden Befehl. Gegebenenfalls ist die Anweisung mehrfach zu wiederholen. Neben den MOV-Befehlen läßt sich die Segment-Override-Technik auch bei anderen Anweisungen verwenden. In den betreffenden Abschnitten findet sich dann ein Hinweis.

## 2.4 Der PUSH-Befehl

Dieser Transferbefehl speichert den Inhalt von 16-Bit-Registern auf dem Stack ab. 8-Bit-Register lassen sich nicht speichern, vielmehr muß das jeweilige 16-Bit-Register benutzt werden. Bild 2.14 zeigt den Ablauf beim PUSH-Befehl.

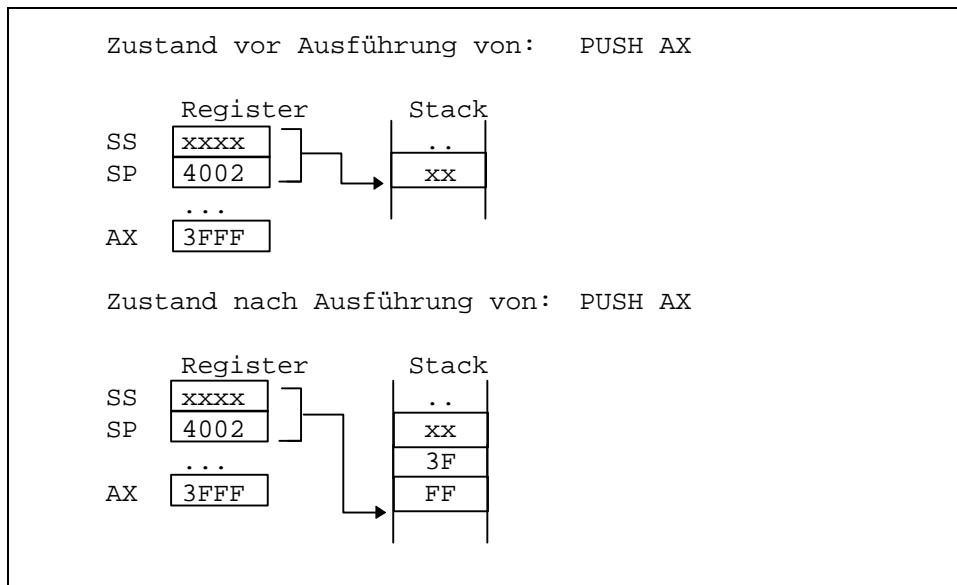


Bild 2.14: Auswirkungen des PUSH-Befehls

Das Register SS adressiert das Segment in dem der Stackbereich liegt. Der Stackpointer (SP) zeigt immer auf das zuletzt auf dem Stack gespeicherte Element. Vor Ausführung des PUSH-Befehls wird der Stackpointer (SP) um den Wert 2 erniedrigt (decrementiert). Erst dann speichert der Prozessor das 16-Bit-Wort auf den Stack. Dabei steht das Low-Byte auf der unteren Adresse. Diese wird durch die Register SS:SP festgelegt. Dies ist zu beachten, falls der Inhalt des Stacks mit DEBUG inspiziert wird.

Der Befehl besitzt die allgemeine Aufrufsyntax:

#### PUSH Quelle

Als Quelle lassen sich die Prozessorregister oder Speicheradressen angeben. Tabelle 2.8 gibt einige gültig PUSH-Anweisungen wieder.

PUSH Operanden	Beispiel
Register	PUSH AX
Seg. Register	PUSH CS
Speicher	PUSH 30[SI]

Tabelle 2.8: Operanden des PUSH-Befehls

Als Register lassen sich alle 16-Bit-Register (AX, BX, CX, DX, SI, DI, BP und SP) angeben. Weiterhin dürfen die Segmentregister CS, DS, ES und SS benutzt werden. Alternativ lassen sich auch Speicherzellen durch Angabe der Indexregister BX, BP, SI, DI und einem Displacement relativ zum jeweiligen Segment auf dem Stack speichern. Hierbei gelten die gleichen Kombinationsmöglichkeiten wie beim MOV-Befehl. Nachfolgend sind einige gültig PUSH-Befehle aufgeführt.

```
PUSH [BP+DI+30]
PUSH 30[BP][DI]
PUSH [3000]
PUSH [SI]
PUSH CS
PUSH SS
PUSH AX
PUSH DS
```

Bei Verwendung der Register BX, SI und DI bezieht sich die Adresse auf das Datensegment (DS), während mit BP das Stacksegment (SS) benutzt wird.

Der PUSH-Befehl wird in der Regel dazu verwendet, um den Inhalt eines Registers oder einer Speicherzelle auf dem Stack zu sichern. Dabei bleibt der Wert dieses Registers oder der Speicherzelle unverändert. Der PUSH-Befehl beeinflusst auch keinerlei Flags des 8088/8086-Prozessors. Die Sequenz:

PUSH AX  
PUSH BX  
PUSH CX  
PUSH DX

legt eine Kopie der Inhalte aller vier Universalregister auf dem Stack ab. Die Register können dann mit anderen Werten belegt werden. Die Ursprungswerte lassen sich mit dem weiter unten vorgestellte POP-Befehl jederzeit wieder vom Stack zurücklesen.

### 2.4.1 PUSHF, ein spezieller PUSH-Befehl

Mit den bereits vorgestellten PUSH-Anweisungen lassen sich nur die Register des 8086-Prozessors auf dem Stack sichern. Was ist aber mit dem Flag-Register? Ein Befehl:

PUSH Flags

wird der Assembler nicht akzeptieren. Um die Flags auf dem Stack zu speichern, ist die Anweisung:

PUSHF

vorgesehen. Mit PUSHF läßt sich zum Beispiel der Zustand des Prozessors vor Eintritt in ein Unterprogramm retten. Beispiele zur Verwendung des PUSH-Befehls werden im Verlauf der folgenden Kapitel noch genügend vorgestellt.

### 2.4.2 Der POP-Befehl

Der POP-Befehl arbeitet komplementär zur PUSH-Anweisung. Bei jedem Aufruf liest der Prozessor ein 16-Bit-Wort vom Stack in das angegebene Register oder die Speicherzelle zurück. Anschließend wird der Stackpointer um 2 erhöht (incrementiert). Nach der Operation zeigt das Registerpaar SS:SP auf das nächste zu lesende Element des Stacks. Bild 2.15 verdeutlicht die Arbeitsweise des POP-Befehls.

Der alte Wert des Registers AX wird durch den POP-Befehl mit dem Inhalt des obersten Stackelements (hier 3FFFH) überschrieben. Der Stackpointer (SP) zeigt nach Ausführung des Befehls auf das nächste zu lesende Element. Der Programmierer ist dafür verantwortlich, daß die Zahl der POP-Anweisungen nie größer als die Zahl der PUSH-Anweisungen wird. Ein Versuch, mit POP ein Element von einem leeren Stack zu lesen, führt in der Regel zu einem Stacküberlauf und damit zu einem Systemabsturz.

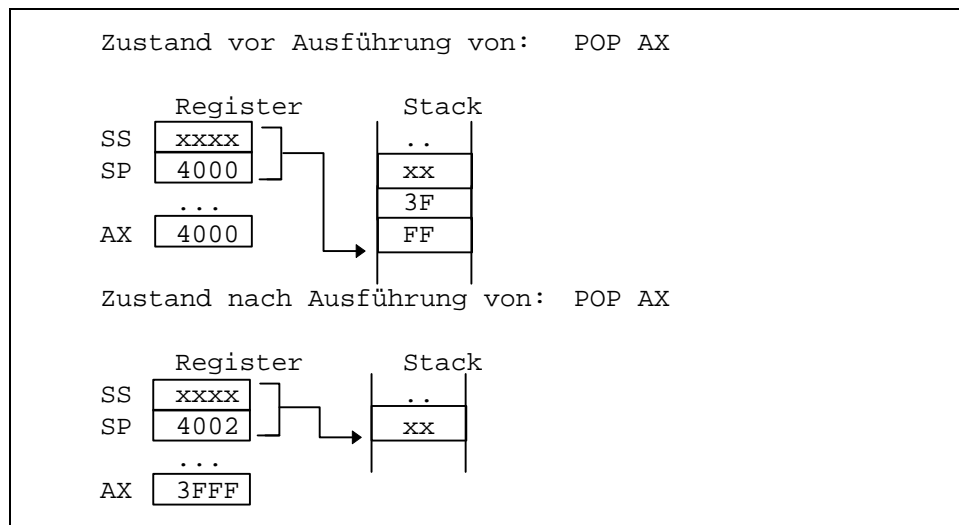


Bild 2.15: Auswirkungen des POP-Befehls

Tabelle 2.9 enthält eine Aufstellung möglicher POP-Befehle.

POP Operanden	Beispiel
Register	POP AX
Seg. Register	POP DS
Speicher	POP 30[SI]

Tabelle 2.9: Operanden des POP-Befehls

Bei der indirekten Adressierung lassen sich Speicherzellen mit dem Inhalt des aktuellen Stackeintrags überschreiben. Hier gelten die gleichen Bedingungen wie beim PUSH-Befehl:

POP BX  
 POP DS  
 POP [0340]  
 POP 30[BX][SI]  
 POP [BP+SI]

Bei der Adressierung über das Register BP (zum Beispiel POP [BP+10]) bezieht sich der Befehl auf das Stacksegment, während bei allen anderen Anweisungen ohne BP (zum Beispiel POP [BX+DI+100]) die Speicherzellen im Datensegment liegen.

Der POP-Befehl darf nicht auf die Register CS, SS und SP angewandt werden, da sonst erhebliche Nebeneffekte auftreten. Betrachten wir einmal das folgende kleine Programm:

```
MOV AX,0000 ; AX = 0
MOV BX,0033 ; BX = 33H
PUSH AX      ; merke AX
....         ; weitere Befehle
....
POP CS       ; lade CS
INT 3
```

Das Programm benutzt einige Register und rettet den Inhalt des AX-Registers. Nach Ausführung verschiedener Befehle wird die Anweisung:

POP CS

ausgeführt. Dadurch tritt ein Seiteneffekt auf: die nächste durch den Prozessor auszuführende Anweisung wird durch die Register CS:IP angegeben. Da CS durch den POP-Befehl verändert wurde, wird die nachfolgende INT 3-Anweisung nicht mehr erreicht, sondern es wird der an der Adresse CS:IP stehende Befehl ausgeführt. Meist handelt es sich aber nicht um ein sinnvolles Programm, so daß ein Systemabsturz die Folge ist. Diese Seiteneffekte sind dem Programm auf den ersten Blick nicht anzusehen. Es ist deshalb grundsätzlich verboten, die Register CS, SS oder SP bei einer POP-Anweisung zu benutzen. Der POP-Befehl verändert den Inhalt des Flag-Registers nicht.

### 2.4.3 Der POPF-Befehl

Ähnlich wie bei PUSHF existiert auch eine eigene Anweisung um die Flags vom Stack zu restaurieren. Der Befehl besitzt die Abkürzung:

POPF

und liest den obersten Wert vom Stack und überschreibt damit den Inhalt des Flag-Registers. Mit der Sequenz:

```
MOV AX,3FFF ; Register Maske
PUSH AX     ; Sichere Maske
POPF        ; setze Flags
```

lassen sich übrigens die Flags definiert setzen. In der Praxis wird man diese Technik allerdings selten anwenden, da meist nur einzelne Bits zu modifizieren sind. Hierfür gibt es spezielle Anweisungen.

Damit möchte ich auf ein kleines Demonstrationsbeispiel unter Verwendung der PUSH- und POP-Befehle eingehen. Ein Programm soll den Inhalt zweier Speicherzellen (DS:150 und DS:152) vertauschen. Dabei darf nur das Register AX zur Speicherung der Zwischenwerte benutzt werden. Nachfolgendes Beispiel zeigt, wie die Aufgabe mit einem Register und den PUSH- und POP-Befehlen zu erledigen ist.



```
MOV AX,[0150]      ; lese ersten Wert
PUSH AX            ; merke den Wert
MOV AX,[0152]      ; lese den 2. Wert
MOV [0150],AX      ; speichere auf 1. Zelle
POP AX            ; hole ersten Wert
MOV [0152],AX      ; setze auf 2. Zelle
```

Versuchen Sie diese Anweisungen mit DEBUG ab Adresse CS:100 zu assemblieren (Start mit A 100) und verfolgen Sie den Ablauf. Der Inhalt der Zelle DS:150 wird gelesen und auf dem Stack zwischengespeichert. Dann ist das Register AX für weitere Werte frei. Nach Umsetzung des Werte von Adresse 152 auf Adresse 150 kann der gespeicherte Wert vom Stack gelesen und unter Adresse 152 eingetragen werden. Es ist aber zu beachten, daß der Programmablauf bei Speicherzugriffen langsamer als bei Registerzugriffen ist. Falls mehrere Register frei sind, sollte im Hinblick auf die Geschwindigkeit auf die Benutzung von PUSH- und POP-Operationen verzichtet werden.

### Programmbeispiel

Nach diesen Ausführungen möchte ich das zweite kleine Demonstrationsprogramm vorstellen. Bei PCs können mehrere parallele Druckerschnittstellen gleichzeitig betrieben werden. Diese Schnittstellen werden unter DOS mit den Bezeichnungen LPT1, LPT2 und LPT3 angesprochen. Manchmal kommt es nun vor, daß ein PC die Ausgänge LPT1 und LPT2 besitzt, an denen jeweils ein Drucker angeschlossen ist (z.B.: LPT1 = Laserdrucker, LPT2 = Matrixdrucker). Dann ist es häufiger erforderlich, daß Ausgaben über LPT1 auf den Drucker an der Schnittstelle LPT2 umgeleitet werden. Über DOS läßt sich zwar die Belegung mittels des Mode-Kommandos umsetzen, aber viele Programme greifen direkt auf die Schnittstelle LPT1 zu. Bestes Beispiel ist ein Bildschirmabzug mit *PrtScr* der auf den Nadeldrucker gehen soll. In der oben beschriebenen Konfiguration wird DOS die Ausgabe immer auf den Laserdrucker leiten. Um auf dem Nadeldrucker den Bildschirmabzug zu erhalten, müssen die Drucker-kabel an den Anschlußports getauscht werden, eine umständliche und nicht ganz befriedigende Möglichkeit.

Hier setzt unser Beispielpogramm an und erlaubt eine softwaremäßige Umschaltung der parallelen Schnittstellen LPT1 und LPT2. Das Programm nutzt die Tatsache, daß das BIOS des Rechners in einem Datenbereich die Zahl der Schnittstellenkarten verwaltet. Der BIOS-Datenbereich beginnt ab Adresse 0000:0400 und umfaßt 256 Byte. Die genaue Belegung ist /1/ aufgeführt. Für unsere Zwecke reicht das Wissen, daß das BIOS in den Adressen:

0000:0408	Portadresse LPT1:
0000:040A	Portadresse LPT2:
0000:040C	Portadresse LPT3:
0000:040E	Portadresse LPT4:

Bild 2.16: Lage der Portadressen

verwaltet. Ist eine Schnittstellenkarte für den betreffenden Anschluß vorhanden, steht ab der betreffenden Adresse die Nummer der I/O-Ports. Fehlt die Schnittstellenkarte, ist die Adresse mit dem Wert 00 00 belegt, d.h. eine Schnittstelle belegt immer 2 Byte. Gegebenenfalls können Sie diese Tatsache selbst mit DEBUG überprüfen. Schauen Sie sich hierzu den Speicherbereich ab 0000:0400 mit dem DUMP-Kommando an. Die 4 seriellen Schnittstellen werden übrigens in gleicher Weise ab der Adresse 0000:0400 verwaltet. Um unser Problem zu lösen, sind lediglich die Einträge für LPT1 und LPT2 in der BIOS-Datentabelle zu vertauschen. Das BIOS wird anschließend die Ausgaben für LPT1 über die physikalische Schnittstelle LPT2 leiten. Ein einfacher aber wirkungsvoller Softwareschalter.

Geben Sie die folgenden Programmanweisungen mit einem Editor in eine Datei mit dem Namen:

#### LPTSWAP.ASM

ein.

```
A 100
;=====
; File: LPTSWAP.ASM (c) G. Born V 1.0
; Aufgabe: Vertausche die Druckerausgänge LPT1
; und LPT2 durch Wechsel der Portadressen im
; BIOS-RAM.
;=====
MOV AX,0000          ; ES := 0 setzen
MOV ES,AX            ;
ES:                  ; Segment Override
MOV AX,[0408]         ; lese 1. Adresse
PUSH AX              ; merke Wert auf Stack
ES: MOV AX,[040A]      ; lese 2. Adresse
ES: MOV [0408],AX      ; speichere an 1. Position
POP AX               ; restauriere 1. Wert
ES: MOV [040A],AX      ; speichere an 2. Position
;
; Programmende
;
MOV AX,4C00           ; DOS-EXIT
INT 21               ;
; hier muß in DEBUG eine Leerzeile folgen

N LPTSWAP.COM
R CX
30
W
Q
```

Listing 2.2: LPTSWAP.COM Programm

Achten Sie bei der Eingabe auf die Leerzeile zwischen der letzten Assembleranweisung und den Steuerbefehlen zur Speicherung des Codes in der COM-Datei (näheres hierzu finden Sie im Kapitel über DEBUG).

Die Quelldatei läßt sich anschließend mit:

```
DEBUG < LPTSWAP.ASM > LPTSWAP.LST
```

übersetzen. Falls keine Fehler auftreten, liegt eine ausführbare COM-Datei vor. Testen Sie diese mit DEBUG aus (DEBUG LPTSWAP.COM). Mit dem 1. Aufruf wird die Belegung der Druckerports vertauscht. Ein zweiter Aufruf stellt wieder den ursprünglichen Zustand her.

Der Aufbau des Programmes ist relativ einfach. Um auf die Adressen im BIOS-Datenbereich zuzugreifen, muß ein Segmentregister mit dem Wert 0000H belegt werden. Denkbar wäre es, hierfür das DS-Register zu benutzen. Da dieses Register aber bei den meisten Programmen in den Datenbereich zeigt, möchte ich hier auf das ES-Register ausweichen. Die Befehle zur indirekten Adressierung benötigen dann zwar einen Segment-Override, was aber hier nicht stört. Sollen andere Druckerausgänge vertauscht werden, lassen sich bei Bedarf die Portadressen im Programm modifizieren (z.B. 0000:040C und 0000:040E für LPT3 und LPT4). Die entsprechenden Einträge werden aus der Tabelle gelesen, per Stack vertauscht und wieder in die Tabelle zurückgeschrieben.

Das Programm besitzt noch eine Besonderheit. In unserem ersten Beispiel wurde der INT 3-Befehl zum Abschluß des Programmes benutzt. Dadurch ließ sich das Beispiel nur unter DEBUG fehlerfrei ausführen. Das vorliegende Programm soll aber später auch als COM-Datei mit der Anweisung:

### LPTSWAP

von DOS aufrufbar sein. Also benötigen wir am Ende des Assemblerprogramms einige Anweisungen, die DOS mitteilen, daß das Programm beendet werden soll. Programme können mit DOS über eine Art Programmbibliothek kommunizieren. Die einzelnen Module lassen sich wie Unterprogramme über den (später noch ausführlicher diskutierten) Befehl INT 21 ansprechen. Die Unterscheidung, welche Teilfunktion der Bibliothek angesprochen werden soll, erfolgt durch den Inhalt des Registers AH. Hier lassen sich Werte zwischen 00H und FFH vom Programm übergeben. Eine genaue Beschreibung der Aufrufschnittstelle der einzelnen Funktionen des INT 21 findet sich in /1/. Im Rahmen dieses Buches werden nur die verwendeten Aufrufe kurz vorgestellt.

### DOS-EXIT

Um ein Programm zu beenden bietet DOS den INT 21-Aufruf DOS-Exit an. Hierzu muß der INT 21 mit dem Wert AH = 4CH aufgerufen werden. In AL kann ein Fehlercode stehen, der sich aus Batchdateien über ERRORLEVEL abfragen läßt. Wird AL = 00H gesetzt, bedeutet dies, das Programm wurde normal beendet. In unseren Assemblerprogrammen wird meist die Sequenz:

```
MOV AX,4C00 ; DOS-Exit  
INT 21
```

auftreten, die das Programm mit dem Fehlercode 0 beendet. DOS übernimmt dann wieder die Kontrolle über den Rechner, gibt den durch das Programm belegten Speicher frei und meldet sich mit den Kommandoprompt (z.B. C>).

Eine verbesserte Version von LPTSWAP.ASM lernen Sie in den folgenden Abschnitten kennen.

#### 2.4.4 Der IN-Befehl

Neben Zugriffen auf den Speicher erlauben die 8086/8088-Prozessoren auch die Verwaltung eines 64 KByte großen Portbereichs. Über diesen Bereich erfolgt dann zum Beispiel die Kommunikation mit Peripherieadaptoren wie Tastatur, Bildschirmkontroller, Floppykontroller, oder den gerade erwähnten parallelen Druckerausgang. Auch wenn Sie nicht allzu häufig direkt auf Ports zugreifen, möchte ich die IN- und OUT-Befehle hier der Vollständigkeit halber beschreiben.

Der IN-Befehl erlaubt es, einen Wert aus dem spezifizierten Port zu lesen. Dabei gilt folgende Befehlssyntax:

```
IN AL,imm8  
IN AX,imm8  
IN AL,DX  
IN AX,DX
```

Mit der Konstanten *imm8* wird eine Port-Adresse im Bereich zwischen 00H und FFH angegeben. Der Befehl liest nun einen 16-Bit-Wert aus dem angegebenen Port aus und speichert das Ergebnis im Akkumulator. Es wird aber noch unterschieden, ob ein 8- oder 16-Bit-Zugriff erfolgen soll.

Die Registerbreite (AX oder AL) spezifiziert dabei, ob ein Wort oder Byte zu lesen ist. Bei den ersten beiden Befehlen wird die Adresse des zu lesenden Ports direkt als 8-Bit-Konstante angegeben. Gültige Befehle sind zum Beispiel:

```
IN AL,0EA      ; lese 8 Bit von  
                ; Port 0EAH in AL  
IN AX,33       ; lese 16 Bit von  
                ; Port 33H in AX
```

Mit einer 8-Bit-Konstanten lassen sich allerdings nur die ersten 255 Ports ansprechen. Vielfach verfügen die Rechner aber über mehr als diese 255 Ports. Daher ist eine Erweiterung der Portadressen auf 16 Bit erforderlich. Diese Adresse läßt sich aber nicht mehr direkt als Konstante beim IN-Befehl angeben. Vielmehr existiert eine Befehlserweiterung, bei der das Register DX zur Aufnahme der 16-Bit-Portadresse

verwendet wird, während der gelesene Wert im Register AX oder AL zurückgegeben wird. Die Breite des Lesezugriffs richtet sich auch hier wieder nach dem angegebenen Register. Nachfolgende Beispiele zeigen, wie der Befehl anzuwenden ist.

```
MOV DX,03FF ; Port 3FF lesen
IN AX,DX    ; als 16 Bit
MOV DX,0000 ; Port 0 lesen
IN AL,DX    ; als 8 Bit
```

Vorher ist die korrekte Portadresse im Register DX zu setzen, da andernfalls undefinierte Ergebnisse auftreten. Der IN-Befehl verändert den Zustand der Flags nicht.

### 2.4.5 Der OUT-Befehl

Der OUT-Befehl bildet das Gegenstück zur IN-Anweisung und erlaubt es, einen Wert an den spezifizierten Port zu übertragen. Dabei gilt folgende Befehlssyntax:

```
OUT imm8,AL
OUT imm8,AX
OUT DX,AL
OUT DX,AX
```

Der zu schreibende Wert ist im Register AX oder AL zu übergeben. Das jeweilige Register spezifiziert, ob ein Wort oder ein Byte zu schreiben ist. Bei den ersten beiden Befehlen wird die Adresse des Ports wieder direkt als 8-Bit-Konstante angegeben. Gültige Befehle sind zum Beispiel:

```
MOV AL,20    ; setze AL
OUT 0EA,AL   ; schreibe Byte auf
              ; Port 0EAH
MOV AX,FFFF  ; setze AX
OUT 33,AX    ; schreibe Wort auf
              ; Port 33H aus AX
```

Mit einer 8-Bit-Konstanten lassen sich ebenfalls nur die Ports mit den Adressen 00H bis FFH ansprechen. Deshalb existiert analog zum IN-Befehl die Möglichkeit, die Adresse indirekt über das Register DX zu spezifizieren. Damit lassen sich 16-Bit-Portadressen zwischen 0000H und FFFFH angeben. Der zu schreibende Wert steht im Register AX oder AL.

```
MOV DX,03FF ; Port 3FF mit
MOV AX,0    ; AX = 0
OUT DX,AX   ; als 16 Bit
              ; beschreiben
OUT DX,AL   ; als 8 Bit
```

; beschreiben

Vor Anwendung des Befehls sind die korrekte Portadresse und der zu schreibende Wert in den Registern DX und AX zu setzen, da andernfalls undefinierte Ergebnisse auftreten. Der OUT-Befehl verändert den Zustand der Flags nicht. Auf Programmbeispiele zu diesem Befehl wird an dieser Stelle verzichtet.

### 2.4.6 Der XCHG-Befehl

Oft ist es erforderlich, den Inhalt zweier Register oder eines Registers und einer Speicherzelle auszutauschen. Mit den bisherigen Kenntnissen über den Befehlssatz läßt sich dies über die folgende Sequenz durchführen:

```
;-----
; tausche den Inhalt AX - BX
;-----
PUSH AX      ; merke AX
MOV AX,BX    ; AX = BX
POP BX       ; BX = AX
```

Zur Lösung dieser einfachen Aufgabe werden mehrere Befehle und ein Zwischenspeicher benötigt. Als Zwischenspeicher kann ein Register oder wie in diesem Beispiel der Stack genutzt werden. Der Zugriff auf den Stack ist aber langsamer als der Zugriff auf die Prozessorregister. Bei Verwendung eines dritten Registers ist ein Wert per MOV in diesem Register zwischenzuspeichern. Häufig ist das Register aber belegt und es sind für die einfache Aufgabe mindestens drei Befehle erforderlich. Um die Lösung zu vereinfachen, besitzt der 8086-Prozessor den XCHG-Befehl (Exchange), mit der allgemeinen Form:

XCHG Destination, Source

Dabei werden die Inhalte von Destination und Source innerhalb eines Befehls vertauscht. Bei Zugriffen auf den Speicher bestimmt die Registergröße ob ein Byte oder ein Wort bearbeitet werden soll. Tabelle 2.10 führt die prinzipiellen Möglichkeiten des XCHG-Befehls auf.

XCHG Operanden	Beispiel
AX, Reg16	CHG AX,BX
Reg8, Reg8	CHG AL,BL
Mem, Reg16	CHG 30[SI],AX

Tabelle 2.10: Operanden des XCHG-Befehls

**Warnung:** Als Operanden dürfen zum Beispiel zwei 16-Bit-Register angegeben werden. Die Segmentregister (CS, DS, SS, ES) lassen sich aber nicht mit dem XCHG-Befehl bearbeiten. Bei den Universalregistern AX bis DX können auch die 8-Bit-

Teilregister (AL bis DH) getauscht werden (XCHG AL,DH). Es ist allerdings nicht möglich, sowohl 16-Bit- als auch 8-Bit-Register zu mischen. Die Anweisung:

XCHG AL,BX

führt deshalb immer zu einer Fehlermeldung.

Bei Zugriffen auf den Speicher per indirekter Adressierung (XCHG [BX],AX) bezieht sich die Adresse im allgemeinen auf das Datensegment. Lediglich bei Verwendung des BP-Registers wird das Stacksegment zur Adressierung benutzt. Bei der indirekten Adressierung lassen sich die gleichen Registerkombinationen wie beim MOV-Befehl benutzen. Das verwendete Register bestimmt dabei, ob ein Wort oder ein Byte zwischen Register und Speicher ausgetauscht wird.

```
XCHG AL,[30FF]      ; tausche Byte
XCHG AX,[30FF]      ; tausche Word
```

Ein Austausch zweier Speicherzellen:

XCHG [3000],[BX]

ist dagegen nicht möglich. Der XCHG-Befehl verändert bei der Ausführung keine Flags.

Ein Programm zur Vertauschung der Registerinhalte AX und BX reduziert sich damit auf folgende Anweisung:

XCHG AX,BX

### Programmbeispiel

Das beim POP-Befehl vorgestellte Beispiel zur Vertauschung der parallelen Schnittstelle läßt sich durch den XCHG-Befehl etwas vereinfachen.

```
A 100
;=====
; File: LPTSWAP.ASM (c) G. Born V 2.0
; Aufgabe: Vertausche die Druckerausgänge LPT1
; und LPT2 durch Wechsel der Portadressen im
; BIOS-RAM. Benutzt den XCHG-Befehl.
;=====
MOV AX,0000          ; ES := 0 setzen
MOV ES,AX            ;
ES: MOV AX,[0408]     ; lese 1. Adresse
ES: MOV BX,[040A]     ; lese 2. Adresse
XCHG AX,BX           ; tausche Adressen
ES: MOV [0408],AX     ; speichere an 1. Position
ES: MOV [040A],BX     ; speichere an 2. Position
;
```

```

; Programmende
;
MOV AX,4C00          ; DOS-EXIT
INT 21              ;
; hier muß in DEBUG eine Leerzeile folgen

N LPTSWAP.COM
R CX
30
W
Q

```

Listing 2.3: LPTSWAP.COM Programm

Das Programm kommt nun ohne Zwischenspeicher (Memory oder Stack) aus.

### Semaphore mit XCHG

Beim XCHG-Befehl sind noch zwei Besonderheiten zu erwähnen. Der erste Punkt betrifft die Realisierung von Semaphoren. Diese werden häufig zur Koordinierung mehrerer Prozesse auf Betriebssystemebene benötigt. Um den Zugang zu einem Drucker für mehrere Prozesse zu verwalten, erhält die Semaphore bei unbelegtem Drucker den Wert 0. Möchte ein Prozeß den Printer belegen, liest er den Wert der Semaphore aus. Falls der Wert ungleich Null ist, hat ein anderer Prozeß den Treiber bereits belegt, der anfragende Prozeß muß warten. Andernfalls setzt der anfragende Prozeß seinerseits die Semaphore auf den Wert 1 und belegt damit den Druckertreiber. Andere Prozesse können erst wieder auf den Treiber zugreifen, wenn der aktive Prozeß die Semaphore zurücksetzt.

Um sicherzustellen, daß während des Lese- und Vergleichsvorgangs auf der Semaphore kein zweiter Prozeß den Wert verändert, darf der Prozeß während dieser Phase nicht unterbrochen werden. Dies ist jedoch nur bei einzelnen Maschinenbefehlen zu garantieren. Hier bietet sich der XCHG-Befehl an, mit der sich folgende Sequenz leicht implementieren läßt:

```

MOV [BX],...    ; Adresse Semaphore
MOV AL,01       ; init Flag
LOCK
XCHG [BX],AL    ; get Semaphore
...             ; falls AL = 1 -> exit
...             ; falls AL = 0 -> weiter, da
                ; Betriebsmittel reserviert

```

Das Register AL wird auf 1 gesetzt und dann wird die Semaphore per XCHG-Befehl gelesen, wobei gleichzeitig der Wert 1 hinterlassen wird. Damit sind andere Prozesse bereits vom Zugang zum Betriebsmittel ausgeschlossen. Falls AL anschließend den Wert 1 besitzt, ist das Betriebsmittel belegt und der Prozeß kann terminieren. Der Wert der Semaphore hat sich ja nicht geändert (er ist nach wie vor 1). War der Wert



= 0, wurde die Semaphore durch den XCHG-Befehl auf 1 gesetzt, der Prozeß hat sich das Betriebsmittel reserviert und kann mit der Bearbeitung beginnen. Die LOCK-Anweisung ist nur bei Multiprozessorsystemen erforderlich. Sie sorgt dafür, daß erst der (aktuelle) Befehl (XCHG) komplett abgearbeitet wird, ehe eine Unterbrechung durch eine andere CPU akzeptiert wird. Damit besteht eine einfache Möglichkeit zur Realisierung von Semaphoren.

### 2.4.7 Der NOP-Befehl

Ein weiterer interessanter Fall tritt auf, falls Quelle und Ziel beim XCHG-Befehl identisch sind. Die Anweisung:

```
XCHG AX,AX
```

weist den Prozessor an, den Inhalt des Registers AX mit sich selbst auszutauschen. Dies bedeutet, daß der Prozessor nichts tun muß. Damit liest er lediglich den Befehl und führt einen Leerschritt aus. Deshalb wird der Befehl allgemein als NOP (No\_Operation) bezeichnet. Der Befehl XCHG AX,AX belegt in der Maschinensprache ein Byte (Opcode 90H). Im Befehlssatz des 8086-Prozessors wurde deshalb die zusätzliche Anweisung:

```
NOP
```

vorgesehen. Der 8086-Assembler generiert aber für die Anweisungen:

```
XCHG AX,AX  
NOP
```

den gleichen Operationscode (90H). Die Ausführung einer NOP-Anweisung hat keinen Einfluß auf den Registerinhalt und beeinflußt auch die Flags nicht. NOP-Befehle werden häufig als Platzhalter in Programmen verwendet.

### 2.4.8 Der XLAT-Befehl

Zur Umcodierung von Werten benutzt man häufig Tabellen. Eine Tabelle enthält dann die zu wandelnden Werte, während die zweite Tabelle an der gleichen Position den neuen Code enthält. Als Beispiel sei die Übersetzung von Umlauten aus dem ASCII-Zeichencode auf die Zeichensätze verschiedener Drucker angeführt. Die ASCII-Zeichen sind gerade so codiert, daß der Wert eines Zeichens zwischen 0 und FFH liegt. Es läßt sich nun eine Tabelle mit 255 Ersatzzeichen angeben. Dort können zum Beispiel alle Kleinbuchstaben durch die Codes der entsprechenden Großbuchstaben ersetzt worden sein. Aufgabe ist es nun, zu jedem eingelesenen Zeichen den entsprechenden Ausgabecode aus der Tabelle zu lesen. Nachfolgendes kleine Programm skizziert, wie diese Aufgabe zu erledigen ist.

```

;-----
; Übersetzungstabelle
; AL = Zeichen, AH = 0
; BX = Zeiger auf Tabelle
; SI = Zeiger in Tabelle
;-----
MOV AH,0          ; clear AH
MOV AL,Zeichen    ; lese Zeichen
MOV BX,Table      ; lese Zeiger
MOV SI,AX         ; Offset
MOV AL,[BX+SI]    ; lese Tabelle
....

```

Das Beispiel läßt sich mit DEBUG nicht nachvollziehen, da diese Programm keine Variablen unterstützt. Das Register BX wird in obigem Beispiel als Basiszeiger auf den Tabellenanfang genutzt. Der Wert des Zeichens in AL dient als Offset in die Tabelle *table*. Der Zugriff kann aber nur über das Register SI oder DI erfolgen. Deshalb ist der Wert in das Register SI zu kopieren. Erst dann kann über die indirekte Adressierung der Eintrag der Tabelle in AL zurückgelesen und weiterverarbeitet werden. Da der Wert des ASCII-Zeichens identisch mit dem Index in die ASCII-Tabelle ist, kann hier auf die explizite Angabe der Tabelle mit den zu wandelnden Zeichen verzichtet werden. Die Umsetzung ist trotzdem recht aufwendig, mit dem XLAT-Befehl läßt sich die Aufgabe wesentlich einfacher erledigen. Bild 2.17 gibt die Wirkungsweise des Befehls schematisch wieder.

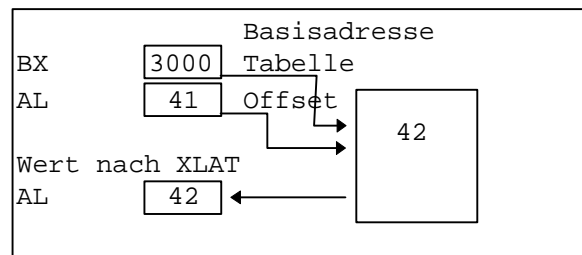


Bild 2.17: Translate Code per XLAT-Befehl

Das Register BX dient als Zeiger auf eine Tabelle mit 255 Einträgen (Bytes). Als Segment wird dabei das Datensegment benutzt. Der Inhalt des Registers AL bezeichnet den Offset in die Tabelle. Der XLAT-Befehl liest den durch BX+AL adressierten Tabellenwert aus dem Datensegment (DS) und speichert diesen im Register AL. In Bild 2.17 wird der Wert 41H aus AL durch den Tabellenwert 42H ersetzt. Obiges Beispiel reduziert sich damit auf folgende Befehle:

```

;-----
; Übersetzungstabelle
; AL = Zeichen, AH = 0
; BX = Zeiger auf Tabelle
;-----
MOV AL,Zeichen ; lese Zeichen
MOV BX,Table   ; lese Zeiger
XLAT           ; hole Zeichen

```

Wichtig ist letztlich nur, daß vor Benutzung der Anweisung das Register BX mit der Anfangsadresse der Tabelle geladen wurde. Der Befehl ist allerdings auf die Bearbeitung von Tabellen mit maximal 255 Byte begrenzt. Die Flags des Prozessors werden bei der Ausführung von XLAT nicht verändert.

### Programmbeispiel

Als weitere einfache Anwendung des XLAT-Befehls möchte ich nun ein Unterprogramm (Teilprogramm) zur Umwandlung einer Hexadezimalziffer (0..F) in die betreffenden ASCII-Zeichen ('0'..'F') vorstellen.

```
A 1000
;=====
; Unterprogramm zur HEX-ASCII-Wandlung
; AL -> Hexziffer, Return: AL -> Zeichen
;=====
; Codetabelle mit den 16 ASCII-(Hex)-Ziffern
DB "0123456789ABCDEF"
; Start ab Adresse 1010 ! für DEBUG
AND AL,0F      ; high nibble Ziffer löschen
MOV BX,1010    ; Startadresse Tabelle
PUSH DS        ; merke Datensegment
PUSH CS        ; DS := CS
POP DS
XLAT           ; konvertiere
POP DS        ; altes Datensegment holen
RET           ; Ende Unterprogramm
```

*Listing 2.4: HEX-ASCII-Konvertierung*

Das Programm benutzt eine Tabelle mit den 16 ASCII-Zeichen der Hexziffern. Diese Tabelle wird in DEBUG mit der DB-Anweisung aufgebaut. Da die Tabelle die ersten 16 Byte belegt, beginnt der eigentliche Programmcode erst ab der Adresse 1010. Die AND-Anweisung (Beschreibung siehe unten) stellt sicher, daß wirklich nur der Code einer Hexziffer im Wertebereich zwischen 0 und 0FH in AL übergeben wird. Nach der Ausführung des Befehls enthält AL das ASCII-Zeichen der betreffenden Hexziffer.

## 2.4.9 Der Befehl LEA

Der Befehl LEA (Load Effektive Adress) ermittelt die 16-Bit-Offsetadresse einer Speicherstelle. Er besitzt die allgemeine Form:

LEA dest, source

Als *dest* muß ein 16-Bit-Universalregister (AX, BX, ...) angegeben werden. Als *source* ist ein Memory-Operand anzugeben. Das folgende Beispiel zeigt, wie der Befehl zu verwenden ist:

LEA BX,[BP+DI+02]

Der Befehl sieht ähnlich wie die bisher bekannten MOV-Anweisungen aus. Aber während bei der MOV-Anweisung der Inhalt der durch den Zeiger [BP+DI+02] adressierten Speicherzelle nach BX transferiert wird, ermittelt der LEA-Befehl der Wert des Zeigers gemäß Bild 2.18 und speichert das Ergebnis im Zielregister (hier BX).

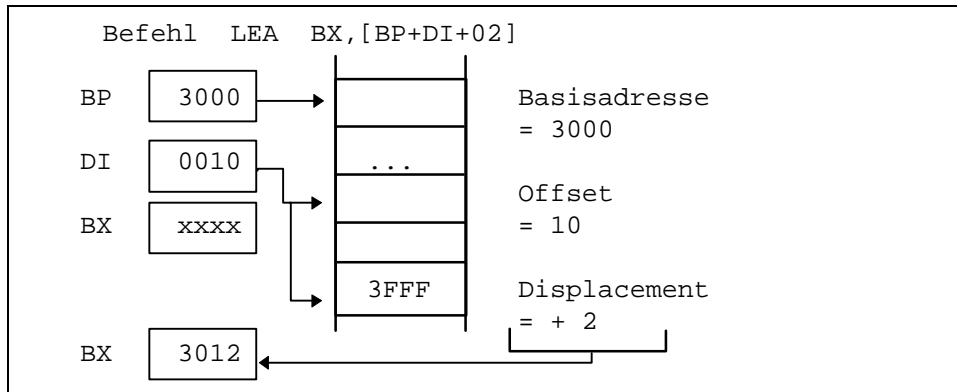


Bild 2.18: Berechnung der effektiven Adresse mit LEA

Wer nach Ausführung des Befehls in BX den Wert 3FFFH erwartet hat, wird enttäuscht sein. Abweichend von MOV greift die Anweisung nicht auf den Speicher zu, sondern ermittelt nur die Summe des in der Klammer [...] angegebenen Ausdrucks. Falls in unserem Beispiel BP den Wert 3000H besitzt, steht nach Ausführung der Anweisung:

```

BP = 3000
DI = 0010
  + 02
BX = 3012
  
```

als Ergebnis der Wert 3012H im Register BX. Der Befehl ist immer dann interessant, wenn der Wert eines Zeigers (z.B. [BP+DI+22]) zu berechnen ist. Ohne die LEA-Anweisung sind mindestens zwei Additionen erforderlich.

#### 2.4.10 Die Befehle LDS und LES

Der Befehl LEA ermittelt nur den 16-Bit-Offset eines Zeigers und legt das Ergebnis in einem Register ab. Oft benötigt ein Programm jedoch 32-Bit-Zeiger. Man denke nur an die Ermittlung des Wertes eines Interruptvektors. Die Adressen der jeweiligen Interruptroutinen liegen beim 8086 auf den Speicherzellen:

0000:0000 - 0000:03FF

Möchte man nun zum Beispiel den Vektor für den Interrupt 0 einlesen, steht dieser auf den Adressen 0000:0000 bis 0000:0003.

Mit dem Befehl LDS (Load Data Segment):

LDS Ziel,Quelle

ist dies leicht möglich. Hierzu ist das Zielregister für den Offset und die Adresse der Quelle anzugeben. Die Anweisung:

LDS SI,[DI]

liest zum Beispiel die Speicherstelle DS:DI und überträgt das Low-Word (Offset) in das Zielregister SI. Das High-Word mit der Segmentadresse wird dann in das DS-Register geladen. Standardmäßig benutzt der LDS-Befehl das Datensegment (DS) zum Zugriff auf den Speicher. Bei der Befehlsausführung wird das High-Word des 32-Bit-Wertes als Segmentadresse interpretiert und immer dem DS-Register zugewiesen. Als Ziel für den Offsetwert darf jedes 16-Bit-Universalregister (AX, BX, ..) angegeben werden. Nachfolgendes kleine Beispiel zeigt, wie der Vektor des INT 0 mit einigen wenigen Befehlen geladen werden kann.

```
;-----  
; laden des INT 0 Vektors  
;-----  
MOV AX,0      ; ES auf Segment  
MOV ES,AX     ; adresse 0000  
ES:           ; Segment Override über ES  
LDS BX,[00]   ; read Vektor 0  
; nun steht der Vektor in  
; DS:BX
```

Hier bleibt noch eine Besonderheit zu erwähnen. Mit der Segment-Override-Anweisung ES: wird die CPU gezwungen, die Adressierung auf dem Quelloperanden nicht über DS:[00] sondern über ES:[00] auszuführen.

Der Befehl *Load Extra Segment (LES)*

LES ziel, quelle

besitzt eine analoge Funktion. Er ermittelt einen 32-Bit-Zeiger und legt den Offsetwert ebenfalls im angegebenen Zielregister ab. Die Segmentadresse wird aber nicht in DS sondern im Extrasegmentregister (ES) gespeichert. Als Zielregister lassen sich die 16-Bit-Universalregister benutzen:

LES DI,[BP+23]

während als Quelle eine Speicheradresse anzugeben ist. Bei Verwendung des BP-Registers im Quelloperanden erfolgt die Adressierung über das Stacksegment SS:[..].

Die Flag-Register des 8086 werden durch diese Befehle nicht beeinflusst.

### 2.4.11 Die Befehle LAHF und SAHF

Diese Befehle wurden im wesentlichen aus Kompatibilitätsgründen zu den 8085-Prozessoren von INTEL eingeführt. Sie erlauben den Austausch des Flag-Registers mit dem Universalregister AH.

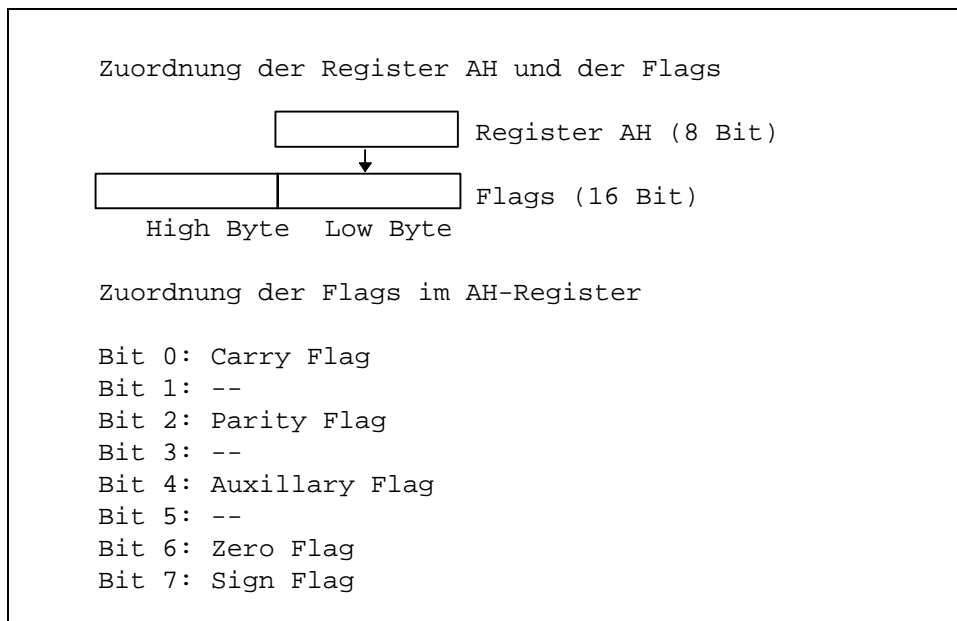


Bild 2.19: Austausch der Flags und AH über die Befehle LAHF und SAHF

Der Befehl *Load AH-Register from Flags*:

#### LAHF

transferiert den Inhalt des 8086-Flag-Registers in das 8-Bit Register AH. Dabei gilt die in Bild 2.19 gezeigte Zuordnung.

Der Befehl *Store AH-Register to Flags (SAHF)* transferiert den Inhalt des AH-Registers zum 8086-Flag-Register. Es gilt dabei die in Bild 2.19 gezeigte Zuordnung. Die beiden Befehle wurden der Vollständigkeit halber aufgeführt. Sie kommen aber in den Beispielprogrammen der folgenden Kapitel nicht vor.

## 2.5 Befehle zur Bitmanipulation

Mit dieser Gruppe von Befehlen lassen sich einzelne Bits oder Gruppen von Bits manipulieren (löschen, setzen, testen). Nachfolgend werden die einzelnen Befehle detailliert besprochen.

### 2.5.1 Der NOT-Befehl

Mit der NOT-Anweisung werden alle Bits des Operanden invertiert (Bild 2.20).

NOT	1001 0111
=>	0110 1000

Bild 2.20: NOT-Operation

Der Befehl besitzt folgende Syntax:

NOT Operator

Als Operator lassen sich dabei Register oder Speichervariable (Byte oder Word) angeben. Tabelle 2.11 enthält eine Zusammenfassung gültiger NOT-Befehle.

NOT AL	Register
NOT AX	Register
NOT [BX+10]	Speicher
NOT [BP+BX]	Speicher

Tabelle 2.11: Die NOT-Befehle

Bei Speicherzugriffen benötigt der Assembler die Schlüsselworte:

NOT WORD PTR [3000]  
NOT BYTE PTR [BX+3]

zur Unterscheidung der Operandengröße. Speichervariable befinden sich in der Regel im Datensegment. Nur bei Verwendung des Registers BP greift die CPU auf das Stacksegment zu. Der Befehl beeinflusst keine Flags.

### 2.5.2 Der AND-Befehl

Eine weitere logische Verknüpfung läßt sich mit dem AND-Befehl durchführen. Dieser besitzt folgende Syntax:

AND Ziel, Quelle

wobei als Operanden Register, Speichervariable und beim Quelloperanden auch Konstante benutzt werden dürfen. Die Datenlänge darf zwischen Bytes und Worten variieren. Quell- und Zieloperanden werden gemäß Bild 2.21 mit der UND-Funktion verknüpft und das Ergebnis findet sich anschließend im Zieloperanden.

	0101 1100
AND	1011 1111
=>	0001 1100

Bild 2.21: AND-Operation

Tabelle 2.12 enthält eine Aufstellung gültiger AND-Befehle.

AND AL,BL	Register/Register
AND CX,[3000]	Register/Memory
AND DL,[BP+10]	Register/Memory
AND [BX+10],0F	Register/Konst.
AND [DI+30],AL	Memory/Register
AND AX,3FFF	Register/Konst.
AND DL,01	Register/Konst.

Tabelle 2.12: Die AND-Befehle

Bei Ausführung des AND-Befehls werden die Flags:

OF, CF

gelöscht und die Flags:

SF, ZF, PF

je nach Inhalt des Zieloperanden modifiziert. Das Auxillary-Flag ist nach der Operation undefiniert. Mit dem AND-Befehl läßt sich zum Beispiel über die Anweisung:

AND AX,AX

prüfen, ob der Inhalt eines Bytes oder eines Wortes den Wert 0 besitzt. Weiterhin lassen sich gezielt einzelne Bits löschen:

AND AL,0F

Die Anweisung löscht die oberen vier Bits des AL-Registers. Solange ein Register als Operand auftritt, bestimmt dieses Register die Operandengröße von Konstanten und



Speichervariablen (z.B. AND AX,[3000]). Bei Zugriffen auf reine Speichervariable benötigt der Assembler die Schlüsselworte:

```
AND WORD PTR [BX+3000], 3000
AND BYTE PTR [BX+SI], 33
```

um den Code für den byte- oder wortweisen Zugriff zu generieren. Zur Adressierung von Speicherzellen (Variable) wird in der Regel das Datensegment (DS) benutzt. Nur bei Verwendung von BP im Adressausdruck erfolgt der Zugriff über das SS-Register.

### Programmbeispiel

Eine praktische Anwendung des AND-Befehls bietet das folgende kleine Programm. Ausgangspunkt hierzu war die Tatsache, daß die NumLock-Taste vieler PCs beim Start eingeschaltet wird. Vor der Benutzung muß der Cursortasten auf dem numerischen Tastenblock deshalb diese Taste manuell abgeschaltet werden, was häufig vergessen wird. Schön wäre es, wenn die NumLock-Taste automatisch beim Systemstart wieder abgeschaltet wird. Diese Aufgabe erledigt das folgende kleine Programm. Wird die Anweisung:

NUMOFF

in die Datei AUTOEXEC.BAT mit aufgenommen, schaltet das Programm die besagte Taste beim Systemstart wieder aus. Geben Sie das Programm mit einem Editor in eine Textdatei (NUMOFF.ASM) ein und übersetzen diese mit DEBUG:

```
DEBUG < NUMOFF.ASM > NUMOFF.LST
```

Anschließend muß eine ausführbare COM-Datei mit dem Namen:

NUMOFF.COM

vorliegen.

Das Programm nutzt wieder Insiderwissen über die Belegung des BIOS-RAM-Bereiches. In der Speicherzelle 0000:0417 speichert das BIOS wie die einzelnen Tasten (NumLock, CapsLock, etc.) gesetzt sind. Die genaue Kodierung ist in /1/ aufgeführt. Ein Bit ist in dieser Speicherstelle für die NumLock-Taste reserviert. Ist es gesetzt, wird die Taste eingeschaltet. Durch Zurücksetzen dieses Bits läßt sich NumLock aber abschalten. Hierfür eignet sich der AND-Befehl hervorragend.

```
A 100
;=====
; File: NUMOFF.ASM (c) Born G. V 1.0
; Aufgabe: Abschalten der NumLock-Taste
;=====
MOV AX,0000      ; ES := 0
MOV ES,AX
```

```

;
; lösche Bit 5 im Tastatur Flag
;
ES: AND BYTE PTR [0417],DF
;
; DOS-EXIT
;
MOV AX,4C00      ; Exit-Code
INT 21
; Leerzeile muß folgen

N NUMOFF.COM
R CX
30
W
Q

```

Listing 2.5: NUMOFF.ASM (Version 1.0)

Hier wird die indirekte Variante des AND-Befehls eingesetzt. Die Konstante DF wird direkt mit der Speicherstelle über:

AND BYTE PTR [0417],DF

verknüpft. Da der Assembler nicht weiß, ob ein Byte oder ein Wort zu bearbeiten ist, muß explizit die Angabe *BYTE PTR* im Befehl auftreten. Die restlichen Befehle sind aus den vorhergehenden Beispielen bereits bekannt. In einem der folgenden Kapitel wird eine erweiterte Version von NUMOFF.ASM vorgestellt, die sich mit einem Assembler übersetzen läßt. Doch für Besitzer von DEBUG besteht mit obigem Listing die Möglichkeit, sich ein nützliches Hilfsmittel zu erzeugen.

Um die NumLock-Taste per Programm einzuschalten, ist lediglich das Bit 5 im BIOS-Datenbereich wieder zu setzen. Hierzu eignet sich der nachfolgend beschriebene OR-Befehl.

### 2.5.3 Der OR-Befehl

Mit der OR-Anweisung läßt sich eine weitere logische Verknüpfung gemäß Bild 2.22 durchführen.

	0111 0000
OR	1010 1001
=>	1111 1001

Bild 2.22: OR-Operation

Der Befehl besitzt die Syntax:

OR Ziel, Quelle

wobei als Operanden Register, Speichervariable und auch Konstante (Quelloperand) benutzt werden dürfen. Die Datenlänge variiert zwischen Bytes und Worten. Das Ergebnis der OR-Operation wird im Zieloperanden gespeichert. Tabelle 2.13 enthält eine Aufstellung gültiger OR-Befehle.

OR AL,BL	Register/Register
OR CX,[3000]	Register/Memory
OR DL,[BP+10]	Register/Memory
OR [BX+10],0F	Register/Konst.
OR [DI+30],AL	Memory/Register
OR AX,3FFF	Register/Konst.
OR DL,01	Register/Konst.

Tabelle 2.13: Die OR-Befehle

Bei Ausführung des Befehls werden die Flags:

OF, CF

gelöscht und die Flags:

SF, ZF, PF

je nach dem Inhalt des Zieloperanden modifiziert. Das Auxillary-Flag ist nach der Operation undefiniert. Mit dem OR-Befehl lassen sich einzelne Bits eines Operanden setzen. Die Anweisung:

OR AH,F0

setzt zum Beispiele die oberen vier Bits des Registers AH. Die Registerbreite eines Operanden bestimmt die Größe des zweiten Operanden bei Speicherzugriffen (z.B. OR AX,[3000]). Bei Zugriffen auf reine Speichervariable benötigen die Assembler die Schlüsselworte:

OR WORD PTR [BX+3000], 3000

OR BYTE PTR [BX+SI],33

Für die Lage der Speichervariablen gelten dabei die üblichen Konventionen zur Benutzung der Segmentregister. Das BP-Register veranlaßt einen Zugriff über das Stacksegment (SS).

## 2.5.4 Der XOR-Befehl

Mit der XOR-Anweisung wird eine Verknüpfung gemäß Bild 2.23 durchgeführt.

	1010 1001
XOR	1001 1011
=>	0011 0010

Bild 2.23: XOR-Operation

Der Befehl besitzt folgende Syntax:

XOR Ziel, Quelle

wobei als Operanden Register, Speichervariable und beim Quelloperanden auch Konstante benutzt werden dürfen. Die Datenlänge variiert zwischen Bytes und Worten und das Ergebnis der XOR-Operation wird im Zieloperanden gespeichert. Tabelle 2.14 enthält eine Aufstellung gültiger XOR-Befehle.

XOR AL,BL	Register/Register
XOR CX,[3000]	Register/Memory
XOR DL,[BP+10]	Register/Memory
XOR [BX+10],0F	Register/Konst.
XOR [DI+30],AL	Memory/Register
XOR AX,3FFF	Register/Konst.
XOR DL,01	Register/Konst.

Tabelle 2.14: Die XOR-Befehle

Bei Ausführung des Befehls werden die Flags:

OF, CF

gelöscht und die Flags:

SF, ZF, PF

je nach dem Inhalt des Zieloperanden modifiziert. Das Auxillary-Flag ist nach der Operation undefiniert. Da der XOR-Befehl alle Bits löscht, die in beiden Operanden den gleichen Wert besitzen, läßt sich den Inhalt eines Registers leicht durch folgende Anweisung löschen:

XOR AX,AX

Der obige Befehl ist wesentlich effizienter als zum Beispiel:

MOV AX,0000

da er weniger Opcodes (Programmcode) benötigt und schneller ausgeführt wird. Die Registerbreite des Zieloperanden bestimmt die Größe des Quelloperanden. Bei Zugriffen auf reine Speichervariablen benötigen die Assembler die Schlüsselworte:

```
XOR WORD PTR [BX+3000], 3000  
XOR BYTE PTR [BX+SI], 33
```

Eine gemischte Verknüpfung von 16- und 8-Bit-Werten ist nicht zulässig. Beim Zugriff auf Speichervariablen gelten die üblichen Konventionen für die Benutzung der Segmentregister. Mit BP als Adressregister erfolgt der Zugriff über SS. Ein XOR-Befehl zwischen zwei Speichervariablen (z.B. XOR [BX],[3000]) ist nicht möglich.

### 2.5.5 Der TEST-Befehl

Der AND-Befehl führt eine logische Verknüpfung zwischen Quell- und Zieloperanden durch. Das Ergebnis wird anschließend im Zieloperanden gespeichert. Dadurch wird aber der ursprüngliche Wert des Zieloperanden zerstört, was oft nicht erwünscht ist. Der 8086-Befehlssatz bietet deshalb die TEST-Anweisung mit folgender Syntax:

TEST Ziel, Quelle

mit Registern, Speichervariablen und Konstanten (Quelle) als Operanden. Die Datenlänge variiert zwischen Bytes und Worten. Der Befehl führt einen AND-Vergleich zwischen den Operanden durch. Allerdings bleibt der Inhalt beider Operanden unverändert, lediglich die Flags:

SF, ZF, PF

werden in Abhängigkeit von der Operation modifiziert. Die Flags:

OF, CF

sind nach der Befehlsausführung gelöscht, während:

AF

undefiniert ist. Mit der Anweisung:

```
TEST AH,01  
JNZ 100
```

prüft der Prozessor ob das Bit 0 in AH gesetzt ist. In diesem Fall wird das Zero-Flag gelöscht. Dadurch wird der folgende Sprungbefehl in Abhängigkeit vom Wert des Bits ausgeführt oder ignoriert. Tabelle 2.15 enthält eine Aufstellung gültiger TEST-Befehle.

TEST AL,BL	Register/Register
TEST CX,[3000]	Register/Memory
TEST DL,[BP+10]	Register/Memory
TEST [BX+10],0F	Register/Konst.
TEST [DI+30],AL	Memory/Register
TEST AX,3FFF	Register/Konst.
TEST DL,01	Register/Konst.

Tabelle 2.15: Die TEST-Befehle

Bei Zugriffen auf reine Speichervariable benötigen die Assembler die Schlüsselworte:

TEST WORD PTR [BX+3000], 3000

TEST BYTE PTR [BX+SI], 33

wobei die üblichen Konventionen für die Benutzung der Segmentregister gelten. Bei Verwendung von BP als Adressregister erfolgt der Zugriff über das SS-Register. Die Benutzung von Operanden mit gemischten Längen (z.B. TEST AX,BL) oder reinen Speichervariablen (z.B. TEST [BX],[300]) ist nicht möglich.

## 2.6 Die Shift-Befehle

Neben den logischen Befehlen zur Bitmanipulation bilden die Shift-Anweisungen eine weitere Befehlsgruppe. Sie ermöglichen Bits innerhalb eines Operanden um mehrere Positionen nach links oder rechts zu verschieben. Die Zahl der Stellen um die verschoben wird, läßt sich entweder als Konstante festlegen (Wert = 1), oder im Register CL angeben. Damit sind Shifts zwischen 1 und 255 Stellen erlaubt. Die Entwickler der CPU haben dabei zwischen arithmetischen und logischen Shiftoperationen unterschieden. Arithmetische Shiftoperationen dienen zur Multiplikation (Shift links) und Division (Shift rechts) des Operanden um den Faktor  $2 * n$ , wobei  $n$  die Zahl der Shiftoperationen angibt. Bei diesem Befehl werden die freiwerdenden Bits mit der Wert 0 belegt. Alternativ existieren die logischen Shift-Befehle. Diese erlauben eine Gruppe von Bits in einem Byte zu isolieren.

Die Flags werden durch die Shift-Befehle in folgender Art beeinflußt:

- ◆ Das Carry-Flag enthält den zuletzt aus dem obersten Bit herausgeschobenen Wert.
- ◆ Das Auxillary-Flag ist immer undefiniert.
- ◆ Die Flags PF, ZF und SF werden in Abhängigkeit vom Wert des Operanden gesetzt.

- ◆ Das Overflow-Flag ist undefiniert, falls mehrfach verschoben wurde. Bei  $n = 1$  wird das OF-Bit gesetzt, falls sich während der Operation das oberste Bit des Operanden (Vorzeichen) geändert hat.

Nachfolgend werden die vier Shift-Befehle der 8086-CPU vorgestellt.

### 2.6.1 Die Befehle SHL /SAL

Die beiden Befehle SHL (Shift Logical Left) und SAL (Shift Arithmetic Left) sind identisch und führen die gleiche Operation durch. Sie besitzen auch die gleiche Syntax:

SHL Ziel, Count

SAL Ziel, Count

Der Inhalt des Zieloperanden (Byte oder Wort) wird um  $n$  Bits nach links verschoben (Bild 2.24).

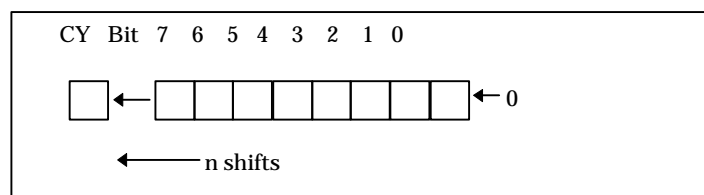


Bild 2.24 : Shift Left bei 8-Bit-Operanden

Dabei bestimmt der Operand *Count* die Anzahl der *Verschiebungen*. Mit:

SHL AX,1

SAL AX,1

wird der Inhalt des AX-Registers um ein Bit nach links verschoben. Um den Inhalt eines Operanden um mehrere Bitpositionen zu verschieben, reicht eine Konstante nicht mehr. Bei Mehrfachverschiebungen muß der Wert im Register CL übergeben werden. Bei der Ausführung der Shift-Anweisung wird auf der rechten Seite Bit 0 bei jeder Shiftoperation zu Null gesetzt. Tabelle 2.16 enthält eine Aufstellung gültiger SHL/ SAL-Befehle.

SAL/SHL AX,1
SAL/SHL AX,CL
SAL/SHL AL,1
SAL/SHL AL,CL
SAL/SHL [DI+1],1
SAL/SHL [DI+1],CL

Tabelle 2.16: Die SAL/SHL-Befehle

Bei Zugriffen auf Speichervariable benötigt der Assembler die Schlüsselworte:

```
SHL BYTE PTR [3000],1
SAL WORD PTR [BX+1],CL
```

um die Länge des zu verschiebenden Operanden zu bestimmen. Bei Speicherzugriffen liegt die Variable standardmäßig im Datensegment. Nur ein Zugriff über BP bezieht sich auf das Stacksegment.

## 2.6.2 Der Befehl SHR

Dieser Befehl (Shift Logical Right) besitzt folgendes Format:

SHR Ziel, Count

und verschiebt den Inhalt des Zieloperanden (Byte oder Wort) um n Bits nach rechts (Bild 2.25).

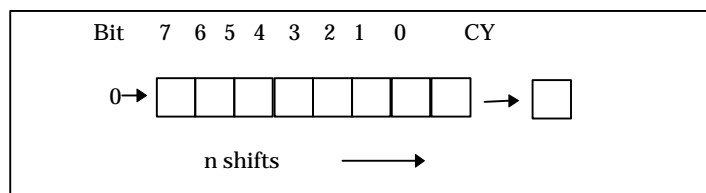


Bild 2.25: Shift Right bei 8-Bit-Operanden

Dabei bestimmt der Operand *Count* die Anzahl der *Verschiebungen*. Bei Mehrfachverschiebungen muß der Zähler im Register CL übergeben werden. Auf der linken Seite des Operanden wird das oberste Bit bei jeder Shiftoperation auf Null gesetzt. Tabelle 2.17 enthält eine Aufstellung gültiger SHR-Befehle.



SHR AX,1
SHR AX,CL
SHR AL,1
SHR AL,CL
SHR [DI+1],1
SHR [DI+1],CL

Tabelle 2.17: Der SHR-Befehl

Bei Zugriffen auf Speichervariablen benötigt der Assembler die Schlüsselwörter:

```
SHR BYTE PTR [3000],1
SHR WORD PTR [BX+1],CL
```

um die Länge des zu verschiebenden Operanden zu bestimmen. Bei Speicherzugriffen liegt die Variable standardmäßig im Datensegment. Nur ein Zugriff über BP bezieht sich auf das Stacksegment.

### 2.6.3 Der Befehl SAR

Dieser Befehl (Shift Arithmetic Right) besitzt folgendes Format:

SAR Ziel, Count

Der Inhalt des Zieloperanden (Byte oder Wort) wird um n Bits nach rechts verschoben (Bild 2.26).

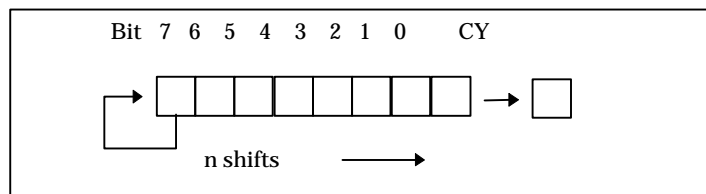


Bild 2.26 : Shift Arithmetic Right (SAR) bei 8-Bit-Operanden

Dabei bestimmt der Operand *Count* die Anzahl der Verschiebungen. Bei einem Shift um mehrere Bits muß das Register CL den Zähler aufnehmen. Im Gegensatz zum SHR-Befehl wird das oberste Bit nicht zu Null gesetzt. Vielmehr bleibt bei jeder Shiftoperation das oberste Bit erhalten und der Wert wird in das nächste rechts stehende Bit kopiert. Dadurch bleibt das Vorzeichen des Operanden erhalten. Tabelle 2.18 enthält eine Aufstellung gültiger SAR-Befehle.

SAR AX,1
SAR AX,CL
SAR AL,1
SAR AL,CL
SAR [DI+1],1
SAR [DI+1],CL

Tabelle 2.18: Der SAR-Befehl

Der SAR-Befehl produziert nicht das gleiche Ergebnis wie ein IDIV-Befehl. Bei Zugriffen auf Speichervariablen benötigt der Assembler die Schlüsselwörter:

```
SAR BYTE PTR [3000],1
SAR WORD PTR [BX+1],CL
```

um die Länge des zu verschiebenden Operanden zu bestimmen. Bei Speicherzugriffen liegt die Variable standardmäßig im Datensegment. Nur ein Zugriff über das BP-Register bezieht sich auf das Stacksegment.

## 2.7 Die Rotate-Befehle

Ähnlich den Shift-Befehlen lassen sich auch die Rotate-Anweisungen zur Verschiebung der Bits innerhalb eines Operanden nutzen. Während bei den Shift-Befehlen aber das *herausgeschobene* Bit verloren geht, bleibt das Bit beim Rotate-Befehl erhalten. Bei den *Rotate through Carry*-Befehlen dient das Carry-Bit als Zwischenspeicher. Ein herausfallendes Bit wird dann im Carry gespeichert, während dessen Inhalt auf der gerade freiwerdenden Bitposition wieder eingespeist wird. Durch dieses Verhalten läßt sich jedes Bit ins Carry bringen und durch relative Sprungbefehle (JC, JNC) testen.

Die Rotate-Befehle beeinflussen einmal das Carry-Flag, welches zur Aufnahme des gerade herausgefallenen Bits dient. Weiterhin wird bei der Rotation um eine Bitposition das Overflow-Flag manipuliert. Wechselt der Wert des obersten Bits, wird OF gesetzt. Dies läßt sich so interpretieren, daß der Rotate-Befehl das Vorzeichen des Operanden verändert hat. Bei Rotate-Anweisungen um mehrere Bitpositionen ist das Overflow-Flag undefiniert.

### 2.7.1 Der ROL-Befehl

Die Anweisung ROL (Rotate Left) rotiert den Inhalt des Operanden (Byte oder Wort) um eine oder mehrere Bitpositionen nach links. Dabei wird links das herausfallende Bit in das Carry-Flag und in Bit 0 kopiert (Bild 2.27).

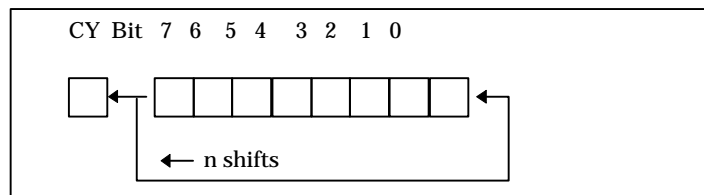


Bild 2.27: Der ROL-Befehl bei 8-Bit-Operanden

Der Befehl besitzt das Format:

ROL Ziel, Count

Count gibt dabei an, um wieviele Bitpositionen der Zieloperand zu rotieren ist. Bei einer Rotation um eine Bitposition läßt sich dies direkt als Konstante im Befehl angeben. Ist der Operand um mehrere Bitpositionen zu rotieren, muß der Rotationszähler im Register CL übergeben werden. Dabei sind Werte zwischen 1 und 255 erlaubt. Tabelle 2.19 enthält eine Aufstellung gültiger ROL-Befehle.

ROL AX,1
ROL AX,CL
ROL AL,1
ROL AL,CL
ROL [DI+1],1
ROL [DI+1],CL

Tabelle 2.19: Die ROL-Befehle

Als Operanden sind Register und Speichervariable erlaubt. Beim Zugriff auf Speicheradressen erwarten einige Assembler die Schlüsselworte:

ROL WORD PTR [3000],1  
ROL BYTE PTR [BX],CL

um die Größe des Operanden zu bestimmen. Speichervariable liegen standardmäßig im Datensegment, daß Stacksegment wird bei Zugriffen über das BP-Register verwendet.

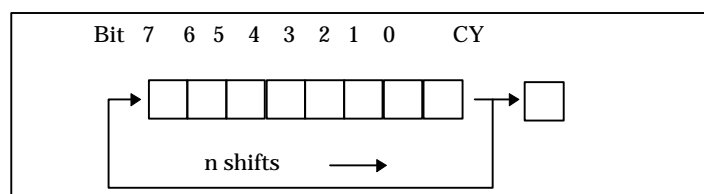


Bild 2.28: Die ROR-Operation bei 8-Bit-Operanden

### 2.7.2 Der ROR-Befehl

Die Anweisung ROR (Rotate Right) arbeitet analog dem ROL-Befehl. Einziger Unterschied: Die Richtung der Rotation ist nach rechts gekehrt (Bild 2.28).

Das Bit 0 des Operanden wird in das Carry-Bit geschoben und in Bit 7 (oder Bit 15) des Operanden kopiert. Die Befehle dürfen sich auf Register und Speichervariable beziehen. Es gelten die üblichen Konventionen zur Verwendung der Segmentregister.

### 2.7.3 Der RCL-Befehl

Die Anweisung RCL (Rotate through Carry Left) verschiebt den Operanden um ein oder mehrere Bitpositionen nach links. Er benutzt dabei das Carry-Bit zur Aufnahme des gerade herausgefallenen Bits (Bild 2.29).

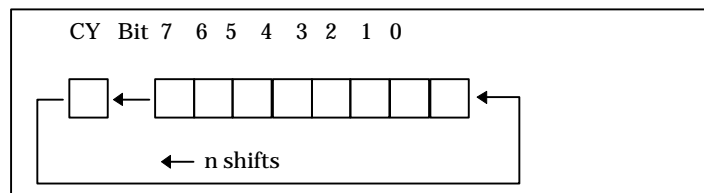


Bild 2.29: Der RCL-Befehl bei 8-Bit-Operanden

Der Befehl besitzt folgendes Format:

RCL Ziel, Count

Mit Ziel wird dabei der Zielooperand (Byte oder Wort) angegeben, der sowohl in einem Register als auch in eine Speicherzelle liegen kann.

RCL AX,1
RCL AX,CL
RCL AL,1
RCL AL,CL
RCL [DI+1],1
RCL [DI+1],CL

Tabelle 2.20: Die RCL-Befehle

Count steht entweder für die Konstante 1 oder für das Register CL und gibt die Zahl der Bitpositionen an, um die zu rotieren ist. Tabelle 2.20 enthält eine Aufstellung gültiger RCL-Befehle.

Beim Zugriff auf Speicheradressen erwartet der Assembler die Schlüsselworte:

```
RCL WORD PTR [3000],1  
RCL BYTE PTR [BX],CL
```

um die Größe des Operanden zu bestimmen. Speichervariablen liegen standardmäßig im Datensegment, das Stacksegment wird bei Zugriffen über das BP-Register benutzt.

### 2.7.4 Der RCR-Befehl

Der Befehl RCR (Rotate through Carry Right) funktioniert analog zum RCL-Befehl. Lediglich die Richtung der Rotation ist nach rechts gerichtet (Bild 2.30).

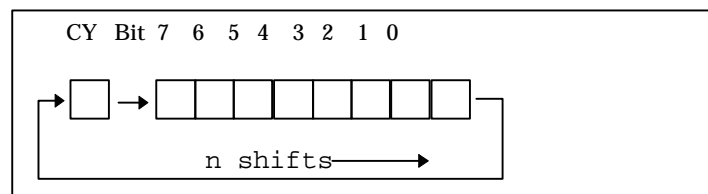


Bild 2.30: Der RCR-Befehl bei 8-Bit-Operanden

Es gilt die gleiche Syntax wie beim RCL-Befehl.

## 2.8 Befehle zur Kontrolle der Flags

Der Befehlssatz des 8086-Prozessors besitzt weiterhin einige Anweisungen um bestimmte Flags definiert zu setzen oder zu löschen. Nachfolgend werden diese Befehle kurz beschrieben.

### 2.8.1 Clear Carry-Flag (CLC)

Die Anweisung CLC (Clear Carry-Flag) besitzt die Syntax:

```
CLC
```

und setzt das Carry-Flag zurück.

### 2.8.2 Complement Carry-Flag (CMC)

Der CMC-Befehl (Complement Carry-Flag) besitzt die Form:

CMC

und liest das Carry-Flag, invertiert den Wert und speichert das Ergebnis zurück.

### 2.8.3 Set Carry-Flag (STC)

Die Anweisung STC (Set Carry-Flag) besitzt das Format:

STC

und setzt das Carry-Flag definiert auf den Wert 1.

### 2.8.4 Clear Direction-Flag (CLD)

Bei den String-Befehlen bestimmt das Direction-Flag die Richtung der Operation. Der Befehl CLD besitzt die Syntax:

CLD

Mit CLD (Clear Direction Flag) wird das Direction-Flag auf 0 gesetzt. Die Indexregister SI/DI werden dann bei jedem Durchlauf automatisch erhöht, die Bearbeitung erfolgt also in Richtung aufsteigende Speicheradressen.

### 2.8.5 Set Direction-Flag (STD)

Die Anweisung STD (Set Direction Flag) setzt das Flag auf den Wert 1. Es gilt die Syntax:

STD

Dadurch wird der Inhalt des Indexregisters SI/DI bei jedem Durchlauf um den Wert 1 erniedrigt. Die Bearbeitung erfolgt also in Richtung absteigender Speicheradressen.

### 2.8.6 Clear Interrupt-Enable-Flag (CLI)

Mit der Anweisung CLI (Clear Interrupt-Enable-Flag) wird das Interrupt-Enable-Flag zurückgesetzt. Der Befehl besitzt die Syntax:

CLI

Dann akzeptiert die 8086-CPU keinerlei externe Unterbrechungen mehr über den INTR-Eingang. Lediglich die NMI-Interrupts werden noch ausgeführt. Dieser Befehl ist wichtig, falls die Bearbeitung von Hardwareinterrupts am PC unterbunden werden soll.

### 2.8.7 Set Interrupt-Enable-Flag (STI)

Mit dem STI-Befehl (Set Interrupt-Enable-Flag) wird die Interruptbearbeitung wieder freigegeben. Der Befehl besitzt die Syntax:

STI

Damit sind die Befehle zur Veränderung der Flags abgehandelt. Beispiele zur Verwendung finden sich in den folgenden Kapiteln.

## 2.9 Die Arithmetik-Befehle

Im Befehlssatz des 8086-Prozessors sind einige Anweisungen zur Durchführung arithmetischer Operationen (Addition, Subtraktion, Multiplikation, Division, etc.) implementiert. Weiterhin finden sich Anweisungen um den Inhalt eines Registers zu incrementieren, zu decrementieren und zu vergleichen. Auf die Befehle des 8087-Prozessors zur Bearbeitung von Fließkommazahlen wird in einem eigenen Kapitel eingegangen.

### 2.9.1 Die Datenformate des 8086-Prozessors

Bevor ich die einzelnen Befehle vorstelle, möchte ich noch kurz auf die Darstellung der verschiedenen Datentypen eingehen. Der 8086-Prozessor kennt vier verschiedene Datentypen:

- ◆ vorzeichenlose Binärzahlen (unsigned integer)
- ◆ Integerzahlen (signed binary)
- ◆ vorzeichenlose gepackte Dezimalzahlen (packed decimals)
- ◆ vorzeichenlose ungepackte Dezimalzahlen (unpacked decimals)

Vorzeichenlose Binärzahlen dürfen verschiedene Längen (8 oder 16 Bit) haben. Solche Zahlen sind bereits im Verlauf des Kapitels aufgetaucht (z.B. MOV

AX,3FFFFH). Integerzahlen werden wie Binärzahlen abgespeichert. Allerdings bilden sie einen positiven und negativen Wertebereich mit 8 oder 16 Bit ab. Um diese Darstellung zu erreichen, wird das oberste Bit der Binärzahl als Vorzeichen interpretiert (Bild 2.31).

$  \begin{array}{r}  \text{FFFF} = 1111\ 1111\ 1111\ 1111 \\  \phantom{FFFF} \quad 0000\ 0000\ 0000\ 0000 \text{ Zweierkomplement} \\  \hline  \phantom{FFFF} \phantom{0000\ 0000\ 0000\ 0000} 1 \\  -1 = -0000\ 0000\ 0000\ 0001  \end{array}  $
---

Bild 2.31: Darstellung einer negativen Zahl

Bei einem gesetzten Bit (z.B. FFFFFH) liegt eine negative Zahl im Zweierkomplement vor. Die Umrechnung einer negativen Zahl in eine positive Zahl (Betragsfunktion) ist im Binärsystem recht einfach. Zuerst wird die Zahl in der Binärdarstellung aufgeschrieben. Dann sind alle Bits zu invertieren (0 wird zu 1 und 1 wird zu 0). Anschließend muß der Wert 1 auf die invertierte Zahl addiert werden. Das Ergebnis bildet den Betrag der negativen Zahl. Die Konvertierung einer positiven Zahl in das negative Äquivalent erfolgt in der gleichen Art. Diese Operationen lassen sich mit der CPU über den NEG-Operator leicht durchführen.

Eine weitere Möglichkeit zur Abbildung von Zahlen bietet das Dezimalsystem. Hier sind nur die Ziffern 0 bis 9 erlaubt. Der Wert 79 wird dann als  $7 \cdot 10 + 9$  interpretiert. Die 80x86-Prozessorfamilie unterstützt mit einigen Befehlen den Umgang mit Dezimalzahlen. Es wird allerdings zwischen gepackter und ungepackter Darstellung unterschieden.

Bei der ungepackten Darstellung dient ein Byte zur Aufnahme einer Ziffer. Die Dezimalzahl 79 läßt sich dann gemäß Bild 2.32 in einem Wort speichern.

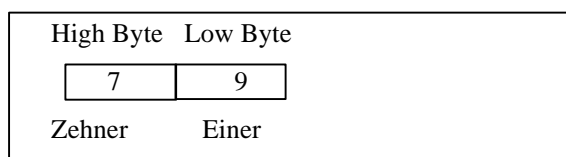


Bild 2.32: Speicherung einer ungepackten Dezimalzahl

In dieser Darstellung ist darauf zu achten, daß der Wertebereich der gespeicherten Zahl in einem Byte maximal bis 9 geht. Die Ziffernfolge 7F ist damit in der Dezimaldarstellung nicht erlaubt.



In Bild 2.32 wird bereits ein Problem deutlich: In einem Byte lassen sich in der Binärdarstellung Werte zwischen 0 und 255 (00 bis FFH) speichern. Bei der Dezimalschreibweise wird der Bereich aber auf die Werte 0 bis 9 (00 bis 09H) beschränkt. Dies ist recht unökonomisch, falls größere Zahlen (z.B. 2379) zu speichern sind. Deshalb existiert die Möglichkeit, Dezimalzahlen in einer gepackten Darstellung zu speichern. Hierzu werden einfach zwei Ziffern in einem Byte untergebracht (Bild 2.33).

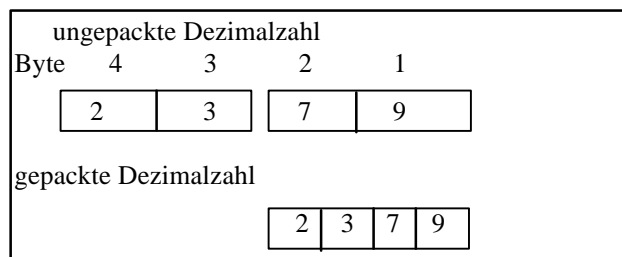


Bild 2.33: Darstellung von gepackten und ungepackten Dezimalzahlen

Ein Byte wird dabei in zwei Nibble zu je 4 Bit aufgeteilt. Die Bits 0 bis 3 nehmen die niederwertige Ziffer auf, während in Bit 4 bis 7 die höherwertige Ziffer steht. Mit vier Bit lassen sich die Zahlen zwischen 0 und 15 darstellen. Die Dezimalschreibweise beschränkt sich allerdings auf die Ziffern 0 bis 9. Die Zahl 33 (dezimal) wird dann gemäß Bild 2.34 kodiert.

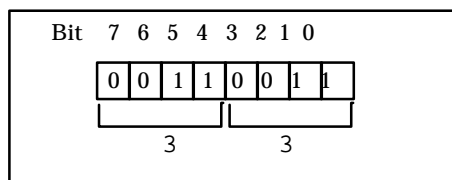


Bild 2.34: Darstellung einer BCD-Zahl

Ein Wert von 3FH ist bei der Darstellung von BCD-Zahlen daher nicht erlaubt. Die gepackte Darstellung von Dezimalzahlen wird häufig als BCD-Darstellung (Binary Coded Decimal) bezeichnet.

Einem Wert läßt es sich in der Regel nicht ansehen, in welchem der oben beschriebenen Zahlenformate er kodiert ist (Tabelle 2.21).

Hex	Binär	unsigned binary	signed binary	unpacked decimal	packed decimal
07	0000 0111	7	+7	7	7
89	1000 1001	137	-119	illegal	89
C5	1100 0101	197	-59	illegal	illegal

Tabelle 2.21: Interpretation eines 8-Bit-Wertes

Die Bewertung liegt allein in den Händen des Programmierers. Die 80x86-Befehle setzen allerdings bestimmte Formate voraus, so daß ein Wert gegebenenfalls in die benötigte Darstellung zu wandeln ist.

## 2.9.2 Der ADD-Befehl

Dieser Befehl ermöglicht die Addition zweier Operanden. Dabei gilt die folgende Syntax:

ADD Ziel, Quelle

Als Operanden sind vorzeichenlose Binärzahlen oder Integerwerte mit 8- oder 16-Bit-Breite erlaubt. Das Ergebnis der Addition wird im Zielooperanden gespeichert. Der Befehl verändert die Flags:

AF, CF, OF, PF, SF, ZF

Als Operanden dürfen Speichervariablen, Register und Konstante (nur Quelle) benutzt werden. Tabelle 2.22 enthält eine Übersicht gültiger ADD-Befehle.

ADD - Befehle	
ADD CX,DX	Register/Register
ADD AL,BH	Register/Register
ADD BX,3FFF	Register/Konst.
ADD BH,30	Register/Konst.
ADD BX,[1000]	Register/Memory
ADD AL,[1000]	Register/Memory
ADD [BX+10],AX	Memory/Register
ADD [BX+10],AL	Memory/Register
ADD [BX+10],3FF	Memory/Konst.
ADD [BX+10],3F	Memory/Konst.

Tabelle 2.22: Die ADD-Befehle

Bei Zugriffen auf Memoryadressen erwarten einige Assembler die Schlüsselworte:

```
ADD WORD PTR [BP+02],0020
ADD BYTE PTR [BX+SI+2],03
```

um zwischen Wort- und Bytewerten zu unterscheiden. Eine Addition zweier Memoryoperanden (z.B. `ADD [BX],[BX+3]`) ist nicht zulässig. Auch eine gemischte Verwendung von Word- und Byteoperanden ist nicht möglich (z.B. `ADD AX,BL`).

Der Befehl läßt sich zum Beispiel verwenden, um eine Dezimalzahl zwischen 0 und 9 in das jeweilige ASCII-Zeichen zu wandeln. Die ASCII-Zeichen 0 bis 9 entsprechen den Hexadezimalzahlen zwischen 30H und 39H. Demnach ist lediglich der Wert 30H zu der Ziffer zu addieren um das entsprechende ASCII-Zeichen zu erhalten. Die läßt sich mit folgender Anweisung bewerkstelligen:

```
ADD AL,30
```

Die Ziffer steht vor Anwendung des Befehls in AL und die Konstante 30H wird addiert. Das Ergebnis findet sich nach Ausführung des Befehls wieder im Register AL.

Bei der indirekten Adressierung (z.B. `ADD AX,[BX+10]`) greift der Befehl über das Datensegment auf den Speicher zu. Nur bei Verwendung des BP-Registers im Adreßausdruck übernimmt die CPU das Stacksegment zur Adressierung. Die Einstellung läßt sich per Segment-Override überschreiben.

### 2.9.3 Der ADC-Befehl

Der Befehl ADC (Add with Carry) addiert analog zu ADD den Wert zweier Operanden und legt das Ergebnis im Zieloperanden ab. Falls das Carry-Flag vor Ausführung des Befehls gesetzt war, wird zusätzlich der Wert 1 addiert. Dadurch läßt sich ein eventueller Übertrag aus einer bisherigen Addition (siehe unten) berücksichtigen.

Der ADC-Befehl besitzt die Syntax:

ADC Ziel, Quelle

Als Operanden lassen sich vorzeichenlose und vorzeichenbehaftete 8- und 16-Bit-Werte verarbeiten. Dabei sind sowohl Register, Konstante (nur Quelle) und Speichervariable als Operanden erlaubt. Die Anweisung setzt folgende Flags:

AF, CF, OF, PF, SF, ZF

Tabelle 2.23 gibt eine Übersicht über gültige ADC-Befehle.

ADC - Befehle	
ADC CX,DX	Register/Register
ADC AL,BH	Register/Register
ADC BX,3FFF	Register/Konst.
ADC BH,30	Register/Konst.
ADC BX,[1000]	Register/Memory
ADC AL,[1000]	Register/Memory
ADC [BX+10],AX	Memory/Register
ADC [BX+10],AL	Memory/Register
ADC [BX+10],3FF	Memory/Konst.
ADC [BX+10],3F	Memory/Konst.

Tabelle 2.23: Die ADC-Befehle

Bei Zugriffen auf Memoryadressen erwarten einige Assembler die Schlüsselworte:

```
ADC WORD PTR [BP+02],0020
ADC BYTE PTR [BX+SI+2],03
```

um zwischen Wort- und Bytewerten zu unterscheiden. Eine Addition zweier Memoryoperanden (z.B. ADC [BX],[BX+3]) ist nicht zulässig. Auch eine gemischte Verwendung von Word- und Byteoperanden ist nicht möglich (z.B. ADC AX,BL). Bei der indirekten Adressierung wird standardmäßig das Datensegment zum Zugriff auf die Speicherzellen benutzt. Eine Ausnahme bildet das BP-Register, welches auf das Stacksegment zugreift.

### Programmbeispiel

Nachfolgendes Beispielprogramm verwendet die beiden Befehle ADD und ADC um zwei 32-Bit-Zahlen zu addieren. Die Werte sind in den Registerpaaren DX:AX und CX:BX zu übergeben. Im ersten Schritt werden die unteren 16 Bit der Operanden (AX + BX) addiert. Das Teilergebnis findet sich im Register AX. Der ADC-Befehl sorgt nun für die Addition der beiden oberen 16-Bit-Werte (DX + CX). Wurde das Carry-Flag bei der Addition von AX + BX gesetzt, berücksichtigt der ADC-Befehl diesen Überlauf automatisch und erhöht das Ergebnis um eins.

```

;=====
;      >>>>  ADD
; Addition: DX:AX = DX:AX + CX:BX (32 Bit)
;=====
; ADD:
ADD AX,BX      ; addiere Low Word
ADC DX,CX      ; addiere High Word mit Carry
RET           ; Exit
;
```

Listing 2.6: Addition

Das Ergebnis der Addition steht anschließend in den Registern DX:AX. Die RET-Anweisung zum Abschluß des Programmes sorgt für dessen Beendigung und wird in einem der nachfolgenden Abschnitte besprochen.

Die Befehle ADD und ADC benutzen in der Regel Binärzahlen als Operanden. Es ist aber durchaus möglich zwei BCD-Werte mit einer solchen Anweisung zu addieren.

### 2.9.4 Der DAA-Befehl

Bei der Addition von gepackten BCD-Zahlen tritt das Problem auf, daß Zwischenergebnisse ungültige BCD-Ziffern aufweisen. Versuchen Sie einmal die zwei BCD-Zahlen 39 und 12 binär zu addieren.

```
  39
+ 12
---
  4B
```

Das Ergebnis 4BH ist keine gültige BCD-Zahl mehr und muß korrigiert werden. Für diesen Zweck existiert der DAA-Befehl (Dezimal Adjust for Addition), der nach der Addition zweier gültiger gepackter BCD-Zahlen eingesetzt wird. Ist der Wert eines Nibble größer als 9, addiert die CPU den Wert 6 hinzu und führt einen Übertrag aus. Obige Rechnung wird dann folgendermaßen ausgeführt:

```
  39
+ 12
---
  4B
+ 06
---
  51
```

womit das Ergebnis wieder eine korrekte BCD-Zahl darstellt. Der DAA-Befehl bezieht sich nur auf den Wert des AL-Registers und besitzt folgende Syntax:

DAA

Er beeinflußt die Flagregister:

AF, CF, PF, SF, ZF

während das OF-Flag undefiniert bleibt.

Das folgende kleine Beispielprogramm benutzt dieses Wissen und addiert die zwei in obigem Beispiel verwendeten gepackten BCD-Zahlen.

```
;-----
; BCD-Addition mit ADD
; und DAA
;-----
```

```
MOV AL, 39    ; 1. BCD-Zahl laden
MOV BL, 12    ; 2. BCD-Zahl laden
ADD AL, BL    ; addiere Werte
;
; 39H 1. BCD-Zahl
; 12H 2. BCD-Zahl
; -----
; 4BH => ungültige BCD-Zahl !!!
;
DAA           ; Korrektur der Ziffern
;
; Das Ergebnis in AL = 51H ->
; ist eine korrekte BCD-Zahl
;
```

*Listing 2.7: BCD-Addition*

Vielleicht vollziehen Sie dieses Beispiel mit DEBUG nach.

### 2.9.5 Der AAA-Befehl

Ähnlich dem DAA-Befehl wird die AAA-Anweisung (ASCII-Adjust for Addition) bei der Addition von ungepackten BCD-Zahlen eingesetzt. Der Befehl korrigiert den Inhalt des Registers AL in eine gültige ungepackte BCD-Zahl. Dabei wird das obere Nibble einfach zu Null gesetzt. Der Befehl besitzt die Syntax:

AAA

und beeinflusst die Flags:

AF, CF

während OF, PF, SF und ZF nach der Ausführung undefiniert sind.

### 2.9.6 Der SUB-Befehl

Mit dem SUB-Befehl lassen sich zwei Binärzahlen subtrahieren. Der Befehl besitzt folgende Syntax:

SUB Ziel, Quelle

und subtrahiert den Quelloperanden vom Zieloperanden, wobei anschließend das Ergebnis im Zieloperanden abgelegt wird. Dabei verändern sich die Flags:

AF, CF, OF, PF, SF, ZF

Als Operanden dürfen 8- oder 16-Bit-Werte als Speichervariable, Register und Konstante (nur Quelle) benutzt werden. Tabelle 2.24 enthält eine Übersicht gültiger SUB-Befehle.

ADC - Befehle	
SUB CX,DX	Register/Register
SUB AL,BH	Register/Register
SUB BX,3FFF	Register/Konst.
SUB BH,30	Register/Konst.
SUB BX,[1000]	Register/Memory
SUB AL,[1000]	Register/Memory
SUB [BX+10],AX	Memory/Register
SUB [BX+10],AL	Memory/Register
SUB [BX+10],3FF	Memory/Konst.
SUB [BX+10],3F	Memory/Konst.

Tabelle 2.24: Die SUB-Befehle

Bei Zugriffen auf Memoryadressen erwarten einige Assembler die Schlüsselworte:

```
SUB WORD PTR [BP+02],0020
SUB BYTE PTR [BX+SI+2],03
```

um zwischen Wort- und Bytewerten zu unterscheiden. Eine Subtraktion zweier Memoryoperanden (z.B. SUB [BX],[BX+3]) ist nicht zulässig. Auch eine gemischte Verwendung von Word- und Byteoperanden ist nicht möglich (z.B. SUB AX,BL).

Bei der indirekten Adressierung greift der Befehl standardmäßig auf das Datensegment zu. Nur bei Verwendung des BP-Registers übernimmt die CPU das Stacksegment zur Adressierung. Die Einstellung lässt sich per Segment-Override überschreiben.

### 2.9.7 Der SBB-Befehl

Der Befehl SBB (Subtract with Borrow) subtrahiert analog zu SUB den Wert des Quelloperanden vom Zieloperanden und legt das Ergebnis im Zieloperanden ab. Falls das Carry-Flag vor Ausführung des Befehls gesetzt war, wird zusätzlich der Wert 1 subtrahiert. Dadurch lässt sich ein eventueller Übertrag aus einer vorhergehenden Operation berücksichtigen (siehe nachfolgendes Beispiel).

Der SBB-Befehl besitzt die Syntax:

SBB Ziel, Quelle

Als Operanden lassen sich vorzeichenlose und vorzeichenbehaftete 8- und 16-Bit-Werte verarbeiten. Dabei sind sowohl Register, Konstante (nur Quelle) und Speichervariable als Operanden erlaubt. Die Anweisung setzt folgende Flags:

AF, CF, OF, PF, SF, ZF

Tabelle 2.25 enthält eine Übersicht über gültige SBB-Befehle.

SBB - Befehle	
SBB CX,DX	Register/Register
SBB AL,BH	Register/Register
SBB BX,3FFF	Register/Konst.
SBB BH,30	Register/Konst.
SBB BX,[1000]	Register/Memory
SBB AL,[1000]	Register/Memory
SBB [BX+10],AX	Memory/Register
SBB [BX+10],AL	Memory/Register
SBB [BX+10],3FF	Memory/Konst.
SBB [BX+10],3F	Memory/Konst.

Tabelle 2.25: Die SBB-Befehle

Bei Zugriffen auf Memoryadressen erwarten einige Assembler die Schlüsselworte:

SBB WORD PTR [BP+02],0020

SBB BYTE PTR [BX+SI+2],03

um zwischen Wort- und Bytewerten zu unterscheiden. Eine Subtraktion zweier Memoryoperanden (z.B. SBB [BX],[BX+3]) ist nicht zulässig. Auch eine gemischte Verwendung von Word- und Byteoperanden ist nicht möglich (z.B. SBB AX,BL). Bei der indirekten Adressierung wird standardmäßig das Datensegment benutzt. Als Ausnahme greift die CPU bei Verwendung des BP-Registers auf das Stacksegment zu.

Nachfolgendes kleine Programm benutzt die Befehle SUB und SBB um zwei 32-Bit-Zahlen zu subtrahieren. Die Zahlen sind in den Registerpaaren DX:AX und CX:BX zu übergeben. Im ersten Schritt werden die beiden unteren Worte subtrahiert. Der Befehl SBB subtrahiert die beiden oberen Worte und berücksichtigt einen eventuell aufgetretenen Überlauf.

```

;
;=====
;      >>>>  SUB
;
; Subtraktion: DX:AX = DX:AX - CX:BX (32 Bit)
;=====
; SUB:
SUB AX,BX      ; subtrahiere Low Word
SBB DX,CX      ; subtrahiere High Word mit Borrow

```



```
RET          ; Exit  
;
```

*Listing 2.8: Subtraktion*

Das Ergebnis der Subtraktion findet sich in DX:AX. Vielleicht probieren Sie dieses Beispiel mit DEBUG auszuführen.

### 2.9.8 Der DAS-Befehl

Bei der Subtraktion von gepackten BCD-Zahlen tritt das Problem auf, daß ähnlich wie bei der Addition Zwischenergebnisse ungültige BCD-Ziffern aufweisen. Der Befehl DAS-Befehl (Dezimal Adjust for Subtraction) korrigiert nach einer Subtraktion zweier gepackter BCD-Zahlen den Inhalt des AL-Registers. Der Befehl besitzt die Syntax:

DAS

und beeinflußt die Flagregister:

AF, CF, PF, SF, ZF

während das OF-Flag undefiniert bleibt.

### 2.9.9 Der AAS-Befehl

Ähnlich dem AAA-Befehl wird die AAS-Anweisung (ASCII-Adjust for Subtraction) bei der Subtraktion von ungepackten BCD-Zahlen eingesetzt. Der Befehl wandelt den Inhalt des Registers AL in eine gültige ungepackte BCD-Zahl um. Dabei wird das obere Nibble einfach zu Null gesetzt. Der Befehl besitzt die Syntax:

AAS

und beeinflußt die Flags:

AF, CF

während OF, PF, SF und ZF nach der Ausführung undefiniert sind.

### 2.9.10 Der MUL-Befehl

Mit dem MUL-Befehl lassen sich zwei vorzeichenlose Binärzahlen multiplizieren. MUL besitzt folgende Syntax:

MUL Quelle

und multipliziert den Quelloperanden mit dem Akkumulator. Das Ergebnis findet sich dann in Abhängigkeit von der Breite der zu multiplizierenden Zahlen in folgenden Registern:

- ◆ Ist der Quelloperand ein Byte, wird das Register AX als Zieloperand genutzt und das Ergebnis steht in AH und AL.
- ◆ Bei 16-Bit-Quelloperanden wird AX als Zieloperand verwendet und das Ergebnis steht anschließend in den Registern DX:AX.

Falls bei der Multiplikation die obere Hälfte des Ergebnisses (AH oder DX) ungleich 0 ist, werden die Flags:

CF, OF

gesetzt, andernfalls werden sie gelöscht. Ein gesetztes Flag bedeutet, daß das Ergebnis der Multiplikation mehr Bits als die ursprünglichen Operanden einnimmt. Der Inhalt der Flags AF, PF, SF und ZF ist nach Ausführung des Befehls undefiniert. Als Quelloperanden dürfen Register und Speichervariable verwendet werden. Tabelle 2.26 gibt eine Auswahl gültiger MUL-Befehle an.

MUL - Befehle	
MUL CX	Register 16 Bit
MUL AL	Register 8 Bit
MUL [BX + 1]	Memory 16/8 Bit

Tabelle 2.26: Die MUL-Befehle

Eine Multiplikation mit Konstanten ist nicht vorgesehen. Die 8086-kompatiblen NEC V20 und V30 CPU's besitzen aber solche Befehle.

Bei Zugriffen auf Memoryadressen erwarten einige Assembler die Schlüsselworte:

MUL WORD PTR [BP+02]  
MUL BYTE PTR [BX+SI+2]

um zwischen Wort- und Bytewerten zu unterscheiden. Bei der indirekten Adressierung greift der Befehl auf das Datensegment zu. Nur bei Verwendung von BP übernimmt die CPU das Stacksegment zur Adressierung. Die Einstellung läßt sich aber per Segment-Override überschreiben. Das folgende kurze Programm führt eine Multiplikation zweier 16-Bit-Zahlen durch.

```

;
;=====
;      >    MUL
;
; Multiplikation: DX:AX = AX * BX (16 Bit)

```

```

;=====
; MUL:
MUL BX      ; multipliziere Word
RET         ; Exit
;

```

*Listing 2.9: Multiplikation*

Die Operanden sind in den Registern AX und BX zu übergeben. Nach Ausführung des Programms findet sich in DX:AX das Ergebnis der Multiplikation.

### 2.9.11 Der IMUL-Befehl

Mit dem MUL-Befehl lassen sich nur vorzeichenlose Binärzahlen multiplizieren. Deshalb bietet der 8086 eine eigene Anweisung zur Multiplikation von Integerzahlen. IMUL (Integer Multiply) erlaubt eine Multiplikation zweier vorzeichenbehafteter Binärzahlen. Der Befehl besitzt folgende Syntax:

IMUL Quelle

Als Operanden lassen sich vorvorzeichenbehaftete 8- und 16-Bit-Werte verarbeiten. Ist der Quelloperand ein Byte, wird das Register AL als Zieloperand genutzt und das Ergebnis steht in AH und AL. Bei 16-Bit-Quelloperanden wird AX als Zieloperand verwendet und das Ergebnis steht anschließend in den Registern DX:AX. Falls bei der Multiplikation die obere Hälfte des Ergebnisses (AH oder DX) ungleich 0 ist, werden die Flags:

CF, OF

gesetzt, sonst werden sie gelöscht. Ein gesetztes Bit signalisiert, daß das Ergebnis der Multiplikation mehr Bits als die ursprünglichen Operanden einnimmt. Der Inhalt der Flags AF, PF, SF und ZF ist nach Ausführung des Befehls undefiniert. Als Quelloperanden dürfen Register und Speichervariablen verwendet werden. Tabelle 2.27 gibt eine Auswahl gültiger IMUL-Befehle an.

IMUL - Befehle	
IMUL CX	Register 16 Bit
IMUL AL	Register 8 Bit
IMUL [BX + 1]	Memory 16/8 Bit

*Tabelle 2.27: Die IMUL-Anweisung*

Eine Multiplikation mit Konstanten ist nicht vorgesehen. Bei Zugriffen auf Memoryadressen über indirekte Adressierung (z.B. IMUL [BX+1]) erwarten einige Assembler die Schlüsselworte:

IMUL WORD PTR [BP+02]  
IMUL BYTE PTR [BX+SI+2]

um zwischen Wort- und Bytewerten zu unterscheiden. Bei der indirekten Adressierung greift der Befehl auf das Datensegment zu. Nur bei Verwendung von BP benutzt die CPU das Stacksegment zur Adressierung. Die Einstellung läßt sich per Segment-Override überschreiben.

### 2.9.12 Der AAM-Befehl

Ähnlich dem AAA-Befehl wird die AAM-Anweisung (ASCII-Adjust for Multiply) bei der Multiplikation zweier gültiger ungepackter BCD-Zahlen eingesetzt um das Ergebnis wieder in eine gültige BCD-Zahl zu wandeln. Der Befehl konvertiert den Inhalt einer zweiziffrigen Zahl aus den Registern AH und AL in eine gültige ungepackte BCD-Zahl um. Dabei muß das obere Nibble eines jeden Bytes den Wert Null besitzen. Der Befehl besitzt die Syntax:

AAM

und beeinflußt die Flags:

PF, SF, ZF

während AF, OF und CF nach der Ausführung undefiniert sind.

### 2.9.13 Der DIV-Befehl

Der 8086 kann vorzeichenlose und vorzeichenbehaftete Binärzahlen, sowie BCD-Werte direkt dividieren. Mit dem DIV-Befehl läßt sich der Akkumulator durch eine vorzeichenlose Binärzahl dividieren. DIV besitzt folgende Syntax:

DIV Quelle

Ist der Quelloperand ein Byte, wird das Register AL als Zieloperand genutzt und das Ergebnis steht in AH und AL. Das Register AH enthält dabei den Divisionsrest, während AL das Divisionsergebnis faßt. Bei 16-Bit-Quelloperanden wird der Divisionsrest in DX gespeichert, während AX das Ergebnis der Division enthält. Wird bei der Division der Darstellungsbereich des Zielregisters verlassen (z.B. Division durch 0), dann führt die CPU einen INT 0 (Division by Zero) aus. Die Ergebnisse sind dann undefiniert. Der Inhalt der Flags AF, CF, OF, PF, SF, und ZF ist nach Ausführung des Befehls undefiniert. Als Quelloperanden dürfen Register und Speichervariable verwendet werden. Tabelle 2.28 gibt einige gültige DIV-Befehle an.

DIV - Befehle	
DIV CX	Register 16 Bit
DIV AL	Register 8 Bitt
DIV [BX + 1]	Memory 16/8 Bit

Tabelle 2.28: Die DIV-Befehle

Bei Zugriffen auf Memoryadressen (z.B. DIV [BX]) erwarten einige Assembler die Schlüsselworte:

DIV WORD PTR [BP+02]  
 DIV BYTE PTR [BX+SI+2]

um zwischen Wort- und Bytewerten zu unterscheiden. Der Befehl bezieht sich bei der indirekten Adressierung auf das Datensegment. Nur bei Verwendung des BP-Registers übernimmt die CPU das Stacksegment zur Adressierung. Die Einstellung läßt sich per Segment-Override überschreiben.

Das nachfolgende kleine Programm übernimmt die Division zweier 16-Bit-Werte. Die Werte sind in den Registern AX und BX zu übergeben.

```

;
;=====
;      >>>>  DIV
;
; Division: AX = AX / BX (16 Bit)  DX = Rest
;=====
; DIV:
DIV BX      ; Dividiere Word
RET        ; Exit
;

```

Listing 2.10: Division

Das ganzzahlige Ergebnis der Division findet sich im Register AX, während DX den Divisionsrest enthält.

## 2.9.14 Der IDIV-Befehl

Mit dem IDIV-Befehl läßt sich der Akkumulator durch eine vorzeichenbehaftete Binärzahl dividieren. IDIV besitzt folgende Syntax:

IDIV Quelle

Ist der Quelloperand ein Byte, wird das Register AL als Zieloperand genutzt und das Ergebnis steht in AH und AL. Das Register AH enthält dabei den Divisionsrest, während AL das Divisionsergebnis faßt. Es lassen sich damit Werte zwischen +127

und -128 verarbeiten. Bei 16-Bit-Quelloperanden wird der Divisionsrest in DX gespeichert, während AX das Ergebnis der Division enthält. Es werden Werte im Bereich zwischen +32767 (7FFFH) und -32767 (8001H) bearbeitet. Wird bei der Division der Darstellungsbereich verlassen (z.B. Division durch 0), dann führt die CPU einen INT 0 (Division by Zero) aus. Die Ergebnisse sind dann undefiniert. Der Inhalt der Flags AF, CF, OF, PF, SF und ZF ist nach Ausführung des Befehls undefiniert. Als Quelloperanden dürfen Register und Speichervariable verwendet werden. Tabelle 2.29 gibt einige gültige Befehle an.

IDIV - Befehle	
IDIV CX	Register 16 Bit
IDIV AL	Register 8 Bitt
IDIV [BX + 1]	Memory 16/8 Bit

Tabelle 2.29: Die IDIV-Befehle

Bei Zugriffen auf Memoryadressen erwarten einige Assembler die Schlüsselworte:

IDIV WORD PTR [BP+02]  
IDIV BYTE PTR [BX+SI+2]

um zwischen Wort- und Bytewerten zu unterscheiden. Der Befehl bezieht sich bei indirekter Adressierung auf das Datensegment. Nur bei Verwendung des BP-Registers übernimmt die CPU das Stacksegment zur Adressierung. Die Einstellung läßt sich per Segment-Override überschreiben.

### 2.9.15 Der AAD-Befehl

Ähnlich dem AAM-Befehl wird die AAD-Anweisung (ASCII-Adjust for Division) bei der Division ungepackter BCD-Zahlen eingesetzt. Der Befehl ist vor Ausführung der Division aufzurufen, um eine gültige BCD-Zahl zu erhalten. Der Befehl modifiziert den Inhalt des Registers AL. Dabei muß der Wert von AH = 0 sein, um bei der DIV-Operation ein korrektes Resultat zu erzeugen. Der Befehl besitzt die Syntax:

AAD

und beeinflusst die Flags:

PF, SF, ZF

während AF, OF und CF nach der Ausführung undefiniert sind.

### 2.9.16 Der CMP-Befehl

Um zwei Werte auf gleich, größer und kleiner zu vergleichen, läßt sich die Subtraktion einsetzen. Mit:

$AX := BX - AX$

ist das Ergebnis 0, falls  $BX = AX$  gilt. Ist  $AX$  größer als  $BX$ , dann findet sich anschließend im Register  $AX$  ein negativer Wert. Nachteilig ist allerdings, daß bei der Vergleichsoperation der Inhalt eines Operanden durch die Subtraktion verändert wird. Hier bietet der CMP-Befehl Abhilfe. Die Anweisung besitzt folgendes Format:

CMP Ziel, Quelle

und vergleicht den Inhalt von Ziel mit Quelle. Hierzu wird der Wert des Quelloperanden vom Zieloperanden subtrahiert. Der Wert der Operanden bleibt dabei aber unverändert. Der Befehl setzt lediglich folgende Bits im Flag-Register:

AF, CF, OF, PF, SF, ZF

im Abhängigkeit vom Ergebnis. Sind die Werte von Quelle und Ziel gleich, dann ist das Ergebnis = 0 und das entsprechende Flag wird gesetzt. Der Flag-Zustand kann anschließend durch bedingte Sprunganweisungen überprüft werden.

Der Befehl läßt sich auf Register, Konstante und Speichervariable anwenden (Tabelle 2.30).

CMP BX, DX	; Register Register
CMP DL,[3000]	; Register Memory
CMP [BX+2],DI	; Memory Register
CMP BH,03	; Register Immediate
CMP [BX+2],342	; Memory Immediate
CMP AX,03FF	; Akku Immediate

Tabelle 2.30: Formen des CMP-Befehls

Als Operanden sind sowohl Bytes als auch Worte erlaubt. Ein Vergleich zwischen Worten / Bytes (z.B. CMP AL,3FFFH) ist allerdings nicht zulässig. Auch ein Vergleich zweier Speichervariablen (CMP [BX],[3]) ist nicht möglich. Wird ein Register verwendet, bestimmt dessen Größe automatisch die Breite (Byte, Word) des Vergleichs. Beim Zugriff auf den Speicher ist eine indirekte Adressierung über Register und Konstante, ähnlich wie beim MOV-Befehl, erlaubt. Die Adressen beziehen sich dabei auf das Datensegment. Nur bei Verwendung des BP-Registers liegt die Variable im Stacksegment. Die Einstellung läßt sich durch ein Segment-Override verändern:

ES: CMP AX, [3000]

Einige Assembler erwarten beim indirekten Zugriff auf den Speicher die Schlüsselworte:

CMP WORD PTR [3FFF], 030  
CMP BYTE PTR [BX+SI+10], 02

Die Programme sind gegebenenfalls an diese Nomenklatur anzupassen.

### 2.9.17 Der INC-Befehl

Der INC-Befehl erhöht den Wert des spezifizierten Operanden um 1. Als Operand darf ein Byte oder Wort als vorzeichenloser Binärwert in einem Register oder als Speichervariable angegeben werden. Tabelle 2.31 gibt einige gültige Befehlsformen an.

INC AX	; Register
INC DL	; Register
INC BP	; Register
INC [BX+2]	; Memory
INC [300]	; Memory

Tabelle 2.31 Formen des INC-Befehls

Einige Assembler erwarten jedoch beim Zugriff auf Speicherzellen die Schlüsselworte:

INC WORD PTR [3000]  
INC BYTE PTR [4000]

Damit läßt sich zwischen Bytes und Worten unterscheiden. Der Befehl beeinflußt die Flags:

AF, OF, PF, SF, ZF

Bei Speicheroperanden (z.B. INC [BX+3]) wird standardmäßig auf das Datensegment zugegriffen. Mit dem Register BP im Operanden bezieht sich die Variable auf das Stacksegment. Eine Umdefinition per Segment-Override-Anweisung ist möglich.

Der INC-Befehl läßt sich gut bei der indirekten Adressierung einsetzen, um aufeinanderfolgende Bytes oder Worte zu adressieren. Die nachfolgenden Anweisungen demonstrieren eine Möglichkeit zur Anwendung des Befehls.

MOV BX, 150 ; Adresse Datenstring  
MOV AX, [BX] ; lese 1. Wert



```
INC BX      ; nächster Wert
INC BX      ; "
ADD AX,[BX] ; addiere 2. Werte
INC BX      ; nächster Wert
INC BX      ; "
ADD AX,[BX] ; addiere 3. Wert
```

Das Register BX dient als Zeiger auf die jeweiligen Daten. Es werden drei Werte addiert. Da jeder Wert 16 Bit umfaßt, sind jeweils zwei INC-Anweisungen vor jedem ADD-Befehl erforderlich. Weiterhin wird der INC-Befehl häufig zur Konstruktion von Schleifen verwendet. In den folgenden Kapiteln finden sich noch genügend Beispiele für die Verwendung der Anweisung.

### 2.9.18 Der DEC-Befehl

Dieser Befehl wirkt komplementär zur INC-Anweisung und erniedrigt ein vorzeichenloses Byte oder Word um den Wert 1. Als Operanden sind Register und Speichervariable erlaubt. Tabelle 2.32 gibt einen Überblick über mögliche Variationen.

DEC AX	; Register
DEC DL	; Register
DEC BP	; Register
DEC [BX+2]	; Memory
DEC [300]	; Memory

Tabelle 2.32: Formen des DEC-Befehls

Bei Speicheroperanden erfolgt der Zugriff über das Datensegment. Ausnahme ist eine Adressierung über das Stacksegment, falls das Register BP verwendet wird. Einige Assembler erwarten bei Speicherzugriffen folgende Schlüsselworte:

```
DEC WORD PTR [BX+3]
DEC BYTE PTR [3000]
```

Der Befehl beeinflusst folgende Flags:

AF, OF, PF, SF, ZF

die sich durch bedingte Sprungbefehle auswerten lassen.

#### Programmbeispiel

Nachfolgendes kleine Programm zeigt die Verwendung des Befehls zur Konstruktion einer Schleife.

```

;
; Einsatz des DEC-Befehls zur Konstruktion
; einer Schleife.
;
      MOV CL,5      ; lade Schleifenindex
; Beginn der Schleife
Loop: ...           ; hier stehen weitere
      ...           ; Befehle
      DEC CL        ; Index - 1
      JNZ Loop      ; Schleifenende
      ...

```

*Listing 2.11: Schleifencode*

Das Beispiel läßt sich in DEBUG nur nachvollziehen, falls die Marke *Loop* als absolute Adresse (z.B. JNZ 100) angegeben wird. Weitere Informationen finden sich im Abschnitt über die Sprungbefehle und in der Beschreibung des Programmes DEBUG im Anhang.

### 2.9.19 Der NEG-Befehl

Die Vorzeichenumkehr einer positiven oder negativen Zahl erfolgt durch Anwendung des Zweierkomplements. Mit den Befehlen:

```

MOV AX,Zahl ; lese Wert
NOT AX      ; Bits invertieren
INC AX      ; Zweierkomplement bilden

```

läßt sich dies bewerkstelligen. Der 8086 bietet jedoch den NEG-Befehl, der die Negation eines Wertes direkt vornimmt. Dieser Befehl besitzt folgende Syntax:

NEG Operand

und subtrahiert den Operanden von der Zahl 0. Dies bedeutet, daß der Operand durch Bildung des Zweierkomplements negiert wird. Tabelle 2.33 gibt einige der möglichen Befehlsformen der NEG-Anweisung an.

NEG AX	; Register
NEG DL	; Register
NEG BP	; Register
NEG [BX+2]	; Memory
NEG [300]	; Memory

*Tabelle 2.33: Formen des NEG-Befehls*

Einige Assembler benötigen zum Zugriff auf Speicherzellen die Schlüsselworte:

NEG WORD PTR [BP+3]  
NEG BYTE PTR [3FFF]

um die Speichergröße festzulegen. Es sind sowohl Zugriffe auf Bytes als auf Worte erlaubt. Standardmäßig liegen die Variablen im Datensegment. Bei Verwendung des Registers BP erfolgt der Zugriff über das Stacksegment. Der Befehl NEG beeinflusst folgende Flags:

AF, CF, OF, PF, SF, ZF

die sich mit bedingten Sprungbefehlen testen lassen.

### 2.9.20 Der CBW-Befehl

Zum Abschluß nun noch zwei Befehle zur Konvertierung von Bytes in Worte und Doppelworte. Bei der Umwandlung von Integerwerten von einem Byte in ein Wort und dann in ein Doppelwort muß das Vorzeichen mit berücksichtigt werden. Die 80x86-Prozessoren stellen hier zwei Befehle zur Verfügung.

Die Anweisung *Convert Byte to Word* nimmt den Inhalt des Registers AL und erweitert den Wert mit korrektem Vorzeichen um das Register AH. Nach der Operation liegt das Ergebnis als gültige 16-Bit-Zahl vor. Die Anweisung besitzt die Syntax:

CBW

und verändert keine Flags.

### 2.9.21 Der CWD-Befehl

Die Anweisung *Convert Word to Double Word* nimmt den Inhalt des Registers AX und erweitert den Wert mit korrektem Vorzeichen um das Register DX. Nach der Operation liegt das Ergebnis als gültige 32-Bit-Zahl vor. Der Befehl besitzt das Format:

CWD

und läßt die Flags unverändert.


## 2.10 Die Programmtransfer-Befehle

In den bisherigen Abschnitten wurden, trotz der beschränkten Kenntnisse über den 8086-Befehlssatz, bereits einige kleinere Programme entwickelt. Dabei wurde (im Vorgriff auf diesen Abschnitt) der INT 21 benutzt. Nun ist an der Zeit, den INT-Befehl etwas eingehender zu besprechen. Weiterhin ist eine Schwäche der bisherigen Programme noch nicht aufgefallen, da die Algorithmen recht kurz waren: alle Programme sind linear angelegt und verzichten auf Verzweigungen, Schleifen und Unterprogrammaufrufe. Bei umfangreicheren Applikationen führt jedoch kein Weg an diesen Techniken vorbei.

Deshalb werden in diesem Abschnitt Anweisungen zur Programmablaufsteuerung besprochen. Die Entwickler der 80x86-Prozessoren haben einen umfangreichen Satz an Anweisungen zur Unterstützung von JMP-, CALL- oder INT-Aufrufen implementiert. Auf den folgenden Seiten werden diese Befehle detailliert behandelt. Vielleicht ist dann klarer, was unter absoluten, bedingten und relativen Sprüngen zu verstehen ist und warum bei falscher Anwendung ein Programmabsturz nicht zu vermeiden ist.

### 2.10.1 Die JMP-Befehle

Als erstes möchte ich auf die unbedingten Sprungbefehle der 80x86-Prozessoren eingehen. Sobald ein solcher Befehl ausgeführt wird, verzweigt der Prozessor (unabhängig vom Zustand der Flags) zum angegebenen Sprungziel (Bild 2.35).



```
JMP LABEL  
.  
.  
.  
LABEL: .  
.
```

*Bild 2.35: Die Wirkung des JMP-Befehls*

In Bild 2.35 veranlaßt die JMP LABEL-Anweisung, daß der Prozessor nicht den auf JMP folgenden Befehl ausführt, sondern die Bearbeitung des Programmes ab der Marke LABEL: fortsetzt. Im Gegensatz zu den später behandelten bedingten Sprüngen wird beim JMP-Befehl immer eine Verzweigung zur Zieladresse durchgeführt. Der Befehl wirkt analog der GOTO-Anweisung in BASIC, PASCAL oder FORTRAN. Bei der Programmentwicklung mit DEBUG läßt sich die Anweisung:

```
JMP LABEL
```

allerdings nicht benutzen, da dieses Programm keine symbolischen Adressen verarbeiten kann. Hier muß die absolute Adresse direkt angegeben werden. Den Befehl:

```
JMP NEAR 01239
```

dürfen Sie zum Beispiel in DEBUG jederzeit verwenden.

### Der JMP NEAR-Befehl

In Bild 2.35 wurde die Sprunganweisung nur schematisch gezeigt. Beim 8086 werden jedoch, in Abhängigkeit von der Sprungweite, verschiedene Befehle verwendet. In diesem Abschnitt wird der JMP-NEAR-Befehl behandelt.

Was versteckt sich hinter diesem Begriff und was hat das für Konsequenzen? Betrachten wir nochmals die 8086-Speicherarchitektur. Der Prozessor muß mit seinen 16-Bit-Registern einen Adreßraum von 1 MByte verwalten. Deshalb ist er zur Segmentierung gezwungen, wobei ein Register die Segmentstartadresse angibt. Das zweite Register enthält den Offset auf die Speicherstelle innerhalb des Segmentes. Eine Adresse wird deshalb immer mit 32 Bit in den Registern CS:IP dargestellt. Sprunganweisungen lassen sich jedoch in zwei Gruppen aufteilen:

- ◆ Sprünge innerhalb des aktuellen Segmentes
- ◆ Sprünge über Segmentgrenzen hinaus

Ein Sprung über die Segmentgrenzen (JMP FAR) benötigt demnach immer eine 32-Bit-Adresse als Sprungziel. Dieser Befehl wird im nächsten Abschnitt behandelt.

Wie sieht es aber beim Sprung innerhalb eines Segmentes (JMP NEAR) aus? Der Programmcode steht immer im Codesegment, dessen Anfangsadresse durch das Register CS definiert wird. Die aktuelle Anweisung wird durch den Instruction Pointer (IP) adressiert. Bei der linearen Abarbeitung der Befehle verändert sich nur der Wert des IP-Registers. Ein Sprung innerhalb des Codesegmentes wirkt sich deshalb auch nur auf dieses Register aus, während der Wert von CS gleich bleibt. Als Konsequenz benötigt der Assembler zur Darstellung des JMP-NEAR-Befehls nur ein Opcodebyte und das neue Sprungziel in Form einer 2-Byte-Adresse (Offset). Der Befehl läßt sich daher zum Beispiel folgendermaßen angeben:

```
JMP NEAR 1200
```

Die Anweisung bedingt eine Programmverzweigung innerhalb des Codesegmentes zur Adresse 1200H. Das Schlüsselwort NEAR signalisiert dem Assembler dabei zusätzlich die Sprungweite. Dies ist im Hinblick auf den später behandelten SHORT-Sprung wichtig.

Für den interessierten Leser möchte ich an dieser Stelle noch etwas tiefer auf die Abbildung des Befehls durch den Assembler eingehen. Nehmen wir an, die Anweisung steht im Codesegment ab der Adresse 100H. Der Assembler legt dann dort drei Codebytes ab:

```
CS:0100 E9 FD 10      ; JMP 1200
CS:0103
```

Der Wert E9 (Opcode) signalisiert der CPU, daß es sich hier um eine NEAR-Sprunganweisung handelt. Daran schließt sich ein Wort mit dem Sprungziel an. Hier hätte eigentlich jeder den Wert 1200H erwartet. Offensichtlich steht dort aber der Wert 10FDH. Was hat es nun mit dieser Zahl auf sich? Im ersten Schritt wäre es sicher logisch gewesen, den Offset ab der Segmentstartadresse (hier 1200) als Sprungziel zu codieren. Die Entwickler haben aber, um den Programmcode relocatibel zu halten, eine andere Codierung gewählt. Das Sprungziel wird nicht absolut, sondern relativ zur aktuellen Adresse des IP-Registers angegeben. Zur Bearbeitung des Befehls liest die CPU den ersten Opcode. Anschließend steht fest, daß zum Befehl E9H ein 16-Bit-Displacement (Sprungadresse) gehört. Die CPU liest nun die beiden folgenden Bytes, wobei anschließend der aktuelle Wert des IP-Registers auf die Adresse CS:103 zeigt. Dies ist die Anfangsadresse des folgenden Befehls. Nun decodiert die CPU das eingelesene Wort und berechnet die Zieladresse. In obigem Beispiel soll zur Adresse 1200 im aktuellen Codesegment verzweigt werden. Damit ergibt sich die Distanz zwischen aktueller Adresse und Sprungziel zu:

```
    Ziel  1200
    IP - 0103
Distanz 10FD
```

Genau dieser Wert wurde vom Assembler hinter dem Opcode E9 abgelegt. Die CPU addiert also das gelesene Displacement zum aktuellen Inhalt des IP-Registers und erhält automatisch den neuen Adreßwert. Ist das Displacement größer als 7FFFH, wird das oberste Bit als negatives Vorzeichen interpretiert. Der Sprung erfolgt dann in Richtung des Segmentanfangs, das Ziel liegt demnach in Richtung des Programmanfangs. Damit lassen sich von der aktuellen Adresse alle Ziele bis zur Entfernung von +/-32-KByte anspringen. Es bleibt also festzuhalten, daß das Sprungziel des JMP NEAR-Befehls immer relativ zur aktuellen Adresse codiert wird. Diese auf den ersten Blick etwas merkwürdige Konstruktion besitzt aber einen großen Vorteil. Wird das Programm von Adresse 0100H nach Adresse 1000H verschoben, bleiben alle Sprunganweisungen gültig, da die Sprungziele ja relativ zu den aktuellen Adressen angegeben wurden. Dies gilt allerdings nur solange das Programm nicht neu übersetzt wird, da dann das angegebene Sprungziel in ein neues Displacement umgerechnet wird. Solange das Programm keine absoluten Adressbezüge (z.B. Zugriff auf Daten oder indirekte Sprungbefehle) enthält, ist es frei im Speicher verschiebbar (relocatibel).

Eine weitere offene Frage bezieht sich auf Sprünge am Beginn oder am Ende eines Segmentes (Bild 2.36).

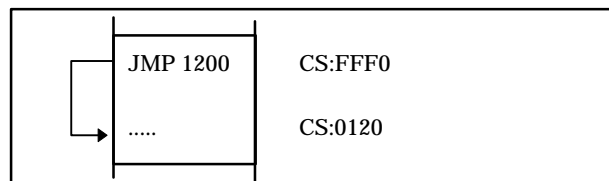


Bild 2.36: Adreßüberlauf beim JMP-Befehl

Was passiert zum Beispiel, falls am Ende des Segments bei Adresse CS:FFF0 ein JMP NEAR 1200 steht? Der Assembler muß das Displacement berechnen, welches den Abstand zwischen aktueller Adresse und Sprungziel beschreibt. Da die Zieladresse am Segmentanfang liegt, errechnet sich der Abstand scheinbar zu:

```
FFF3 Ende JMP-Anweisung
-0120 Sprungziel
FED3
```

Das Ergebnis FED3 ist aber größer als 32 KByte (7FFFH), sofern die Zahl als positive Ganzzahl interpretiert wird und ist damit als Displacement ungültig. Hier muß der Assembler einen anderen Weg wählen. Die Distanz berechnet sich aus den zwei Abständen zu den jeweiligen Segmentgrenzen:

```
10000 Anfang folgendes Segment
- FFF3 Ende JMP-Anweisung
000D Distanz 1
+0120 Distanz 2
012D Displacement
```

Als Displacement errechnet sich ein Wert von 012DH. Die Kontrollrechnung durch Addition des Displacements auf den aktuellen Befehlszähler führt aber scheinbar aus dem Segment heraus zu einer Adresse im folgenden Segment:

```
FFF3 Ende JMP-Anweisung
+ 012D Displacement
1 0120 Sprungziel
```

Ein Sprung aus dem Segment heraus ist aber mit einem JMP NEAR-Befehl nach der Definition unmöglich. Das Ergebnis bestätigt dies auch indirekt: die Zahl 10120H paßt nicht mehr in das 16-Bit-Register IP. Die führende 1 fällt beim Segmentüberlauf heraus und der verbleibende Rest von 0120 entspricht aber genau der gesuchten Sprungadresse. Der Assembler sorgt also dafür, daß immer das korrekte Displacement in einem JMP NEAR-Befehl eingesetzt wird.

Falls Sie jetzt als Einsteiger nur Bahnhof verstanden haben, ist dies nicht weiter tragisch: Sie können die Ausführungen im letzten Abschnitt vorerst ohne Nachteile übergehen, da der Assembler und DEBUG automatisch die Umrechnung der Adressen

vornimmt. Sie geben immer die absoluten Sprungadressen (z.B. `JMP NEAR 1200`) ein. Vielleicht experimentieren Sie trotzdem etwas mit `DEBUG`. Ich habe diese Zusammenhänge hier etwas ausführlicher behandelt, da in der Literatur kaum auf den Aspekt eingegangen wird.

### Programmbeispiel

Nachfolgend möchte ich nun die Verwendung des `JMP`-Befehls an einem weiteren kleinen Beispiel demonstrieren.

```
A 100
;=====
; File : HELLO.ASM
; Demoprogramm zur Stringausgabe in DOS
; in der Version für DEBUG
;=====
JMP NEAR 0119      ; Sprung nach Start
;
;-----
; Datenbereich mit der Textkonstanten
;-----
;
DB "Hallo : Version 1.0",0D,0A,"$"
;
;-----
; Aufruf der INT 21 Funktion 09 zur
; Stringausgabe auf dem Bildschirm.
; Register: AH = 09, DS:DX = Zeiger
;           auf den String
; Der String ist mit dem Zeichen $ ab-
; zuschließen. Hinweis: Bei COM-Files
; gilt immer CS = DS = SS = ES !!!
;-----
; Start:
MOV AH,09          ; DOS-Write String
MOV DX,0103        ; Zeiger auf String
;                 DS = CS !!!
INT 21             ; Text ausgeben
MOV AX,4C00        ; DOS Exit
INT 21
;
; Programm Ende -> Befehle zum Abspeichern
; des Codes in eine COM-Datei

R CX
030
N HELLO.COM
W
Q
```

*Listing 2.12: Das Programm HELLO.ASM*

Das Programm aus Listing 2.5 gibt die Meldung:



Hallo : Version 1.0

auf dem Bildschirm aus. In Listing 2.5 wurde die auszugebende Textkonstante an den Code angehängt. Dies hat aber zu Folge, daß sich die Startadresse des Datenbereiches bei jeder Programmänderung verschiebt, sofern die Daten nicht auf einer absoluten Adresse liegen. Sollen nun Parameter in der COM-Datei durch externe Programme gelesen oder überschrieben (gepatcht) werden (z.B. die Textkonstante), ist dies bei sich ständig ändernden Adressen sehr schwierig. Um das Problem zu umgehen, besteht die Möglichkeit, Konstanten und Daten an den Programmanfang zu legen. Codeänderungen haben dann keinen Einfluß mehr auf die Lage der Daten. MS-DOS lädt den Inhalt einer COM-Datei ab dem Offset 100H in ein reserviertes Codesegment von 64 KByte. In der COM-Datei sind die Texte dann direkt am Fileanfang gespeichert. Genauer zu diesem Thema findet sich in [1]. Allerdings erwartet DOS als erstes einen ausführbaren Befehl im Speicherbereich ab CS:100, so daß dort keine Daten gespeichert werden dürfen. Hier leistet der JMP NEAR-Befehl gute Dienste. Die Anweisung:

JMP NEAR 119

wird als erster Befehl im Programm verwendet und damit ab Offset CS:100 geladen. Er veranlaßt, daß der Prozessor den nächsten Befehl erst ab Adresse CS:0119 ausführt. Da der JMP NEAR-Befehl ja drei Byte umfaßt, lassen sich im Zwischenraum ab Offset 103 die Daten (z.B. der Text) unterbringen. In der COM-Datei findet sich der Text dann immer an einer festen Position (hinter den drei Opcodes des JMP-Befehls) ab Offset 103H und kann bei Bedarf leicht gepatcht werden. Das Programm aus Listing 2.6 läßt sich mit einem Texteditor erstellen und in der Datei HELLO.ASM ablegen.

Falls bei einem Sprung das Sprungziel noch nicht bekannt ist, kann vor der ersten Übersetzung mit DEBUG als Ziel die Konstante 0100 eingetragen werden. Mit der Anweisung:

DEBUG < HELLO.ASM > HELLO.LST

wird die Quelldatei übersetzt, wobei in HELLO.LST ein Listing mit den Adressen und Befehlen angelegt wird. Aus diesem Listing ist dann die gesuchte Adresse des Sprungziels zu ermitteln (hier 119H) und im Quelltext einzutragen. Dann wird die Assemblierung wiederholt, um eine funktionsfähige COM-Datei zu erzeugen. Die Länge des zu speichernden Codebereiches läßt sich ebenfalls aus dem Listfile ermitteln. Weitere Hinweise zum Umgang mit DEBUG finden sich im Anhang. Die Arbeit mit absoluten Sprungadressen ist etwas mühsam, läßt sich bei DEBUG aber nicht vermeiden. Bei Verwendung eines Assemblers übernimmt dieser die Adressberechnung, so daß symbolische Marken verwendbar sind. Entsprechende Beispiele finden sich in den folgenden Kapiteln.

Nach diesen Ausführungen möchte ich noch kurz auf die allgemeine Syntax des JMP-Befehls eingehen. Der Befehl ist gemäß folgender Syntax einzugeben:

### JMP NEAR Ziel

Das Prefix NEAR weist den Assembler an, einen 3-Byte-Befehl für einen Sprung innerhalb des aktuellen Codesegments zu generieren. Das Displacement (Ziel) wird nach den oben besprochenen Regeln berechnet und als Wort abgespeichert. Auf der Assemblerebene läßt sich das Sprungziel sowohl symbolisch als Marke, als absolute Konstante, in einem der 16-Bit-Universalregister, oder indirekt über eine Speicherzelle angeben (z.B. JMP [1200]). Tabelle 2.35 enthält eine Aufstellung der gültigen JMP NEAR-Befehle.

Befehl	Beispiel
JMP NEAR Konst16	JMP NEAR 1200
JMP NEAR Label	JMP NEAR Weiter
JMP NEAR Reg16	JMP NEAR BX JMP NEAR AX JMP NEAR [BX]
JMP NEAR Mem16	JMP NEAR [1200] JMP NEAR [BX+DI] JMP NEAR [BP+10]

Tabelle 2.35: JMP NEAR-Befehle

Bei Sprüngen deren Adresse indirekt aus einem Register oder einer Speicherzelle ermittelt wird, sind allerdings einige Besonderheiten zu beachten. Bei einem Sprung mit absolutem Sprungziel (z.B. JMP NEAR 1200) errechnet der Assembler die Distanz zum Ziel und legt das Displacement im Codebereich mit ab. Dadurch sind relative Sprünge möglich, die Programme werden frei im Speicher verschiebbar. Bei der Adressierung des Sprungziels über Register oder Speicherstellen ist die Zieladresse zur Übersetzungszeit allerdings nicht bekannt. Bei der Anweisung:

### JMP AX

bestimmt der Inhalt des AX-Registers das Sprungziel. Um eine einfache Handhabbarkeit für den Programmierer zu erreichen, erfolgt die Angabe der Zieladresse bei indirekter Adressierung nicht mehr relativ sondern absolut. Die Anweisungen:

```
MOV AX, 1200 ; lade Sprungziel
JMP NEAR AX ; Sprung ausführen
```

veranlassen deshalb eine Programmverzweigung zur absoluten Adresse CS:1200. Damit sind solche Programme allerdings nicht mehr frei im Speicher verschiebbar. Bei der Adressierung über Speicherzellen gelten die gleichen Bedingungen. Die Anweisung:

```
JMP NEAR [BX+DI+3]
```

lädt den Inhalt des durch DS:BX+DI+3 adressierten Speicherwortes in das IP-Register und führt einen Sprung zu dieser Adresse aus. Als Segmentregister für einen Speicherzugriff wird immer DS genutzt. Lediglich beim Zugriff über BP wird das Stacksegment verwendet.

### Der JMP SHORT-Befehl

Bleiben wir weiterhin bei Sprüngen im gleichen Codesegment. Beim JMP NEAR-Befehl wird die Zieladresse mindestens in 2 Byte codiert. In der Praxis kommt es aber häufig vor, daß ein Sprung nur über einige wenige Befehle oder Byte erfolgt. Das Programm in Listing 2.5 steht hier als typisches Beispiel. Der Sprung am Programm-anfang geht nur über den Datenbereich von wenigen Byte. Enthält ein solches Programm viele dieser kurzen Sprünge, ist es recht unökonomisch, jedesmal einen 3-Byte-Befehl einzusetzen. Die Überlegung basiert darauf, daß es möglich sein muß, das Sprungziel mit einem Byte zu beschreiben. Wird die Zieladresse wieder relativ zur aktuellen Programmadresse angegeben, läßt sich mit einem Byte ein Bereich von -128 bis + 127 Byte ansprechen. Mit dem Opcode für den Befehl umfaßt dann eine JMP-Anweisung nur noch 2 Byte, was einer Reduzierung um 30 % entspricht.

Damit sind wir bei einer speziellen Implementierung, dem JMP SHORT-Befehl angelangt. Dieser wird immer dann verwendet, falls sich das Sprungziel mit einem Byte (-128 bis + 127) darstellen läßt. Wird in DEBUG nur der Befehl:

JMP 119

einggegeben, analysiert der Assembler die Distanz zum aktuellen Wert des Instruction Pointer (IP). Bei Werten kleiner 127 generiert er dann einen 2 Byte Sprungbefehl (JMP SHORT). Andernfalls wird automatisch ein 3-Byte-Sprungbefehl (JMP NEAR) verwendet.

Vielleicht stellen Sie sich nun die Frage, warum dann noch die Präfixe NEAR und SHORT notwendig sind, wo doch der Assembler die Sprünge automatisch verwaltet? Leider stecken in der automatischen Generierung des optimalen Befehls zwei Nachteile, die nicht verschwiegen werden sollen. Nehmen wir einmal an, Sie geben ab der Adresse 100 folgenden Befehl ein:

JMP 109

dann kann der Assembler die Distanz berechnen und generiert einen JMP SHORT-Befehl. Sie wollen aber als Anwender nicht immer die absolute Adresse vorgeben, sondern bevorzugen symbolische Marken als Ziel. Dann sollten wir uns einmal folgendes Beispiel im Assembler-Format ansehen:

```
START: MOV AX,0900    ; lade Konstante
        JMP WEITER    ; Fortsetzung
;
; Datenbereich mit dem Textstring
```

```

;
DATEN DS "Hallo $"
;
WEITER: MOV DX, OFFSET DATEN ; Adresse
        INT 21
        JMP START ; Endlosschleife

```

Das Programm enthält eine Endlosschleife und gibt die Meldung "Hallo" auf dem Bildschirm aus. Zwar ergibt dies keinen rechten Sinn, aber die Konstruktion soll ja nur zur Erläuterung der Problematik dienen. Im Programm sind drei Labels enthalten, die den Datenbereich und die Sprungziele definieren. Versetzen wir uns nun in die Lage des Assemblers. Sobald er den Befehl `JMP WEITER` erkennt, soll er entscheiden, welcher Sprungbefehl eingesetzt wird. Da er die Adresse des Labels `WEITER` noch nicht kennt, fehlt ihm jegliche Grundlage zur Berechnung der Distanz. Folglich generiert er einen 3-Byte-Sprungbefehl, da dieser immer paßt. Erst wenn das Label `WEITER` gefunden wird, besteht die Möglichkeit zur Korrektur. Leider verändert sich dann aber die Codelänge, so daß alle eventuell dazwischenliegenden Labels nicht mehr gültig sind. Aus diesem Grund wird vom Assembler bei sogenannten Vorwärtsreferenzen immer der 3-Byte-JMP-NEAR-Opcode eingesetzt. Bei der Anweisung `JMP START` sieht die Sachlage dagegen anders aus. Das Label `START` wurde bereits bearbeitet und der Assembler kann die Distanz berechnen. Daher wird er in obigem Beispiel eine `JMP SHORT`-Anweisung generieren. Die Optimierung klappt also offenbar nur bei Rückwärtsprüngen. Bei vielen Vorwärtsreferenzen kann der Assembler demnach keine `JMP SHORT`-Befehle verwenden. Nun argumentieren viele Programmierer, daß sie mit diesem Sachverhalt leben können. Hier möchte ich aber auf ein weiteres Problem hinweisen, welches bereits in unserem Beispielprogramm auftreten kann. Wird in Listing 2.5 das Prefix `NEAR` beim `JMP`-Befehl weggelassen, generiert `DEBUG` beim Sprung zur Adresse 119 eine `JMP SHORT`-Anweisung. Dieser Befehl umfaßt 2 Byte, wodurch die Daten bereits ab Offset 102 beginnen. Bezieht sich ein Befehl im Programm auf Textkonstante in diesem Bereich, ist die Startadresse relevant. Was passiert aber, falls der Bereich mit den Konstanten während der Programmentwicklung solange vergrößert wird, bis die Sprungweite 127 Byte übersteigt?

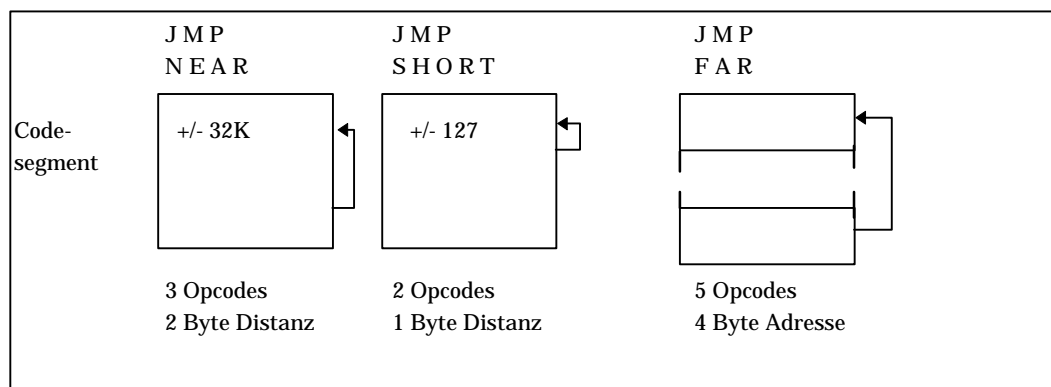


Bild 2.37: Aufstellung der verschiedenen `JMP`-Befehle

Dann setzt der Assembler plötzlich den 3-Byte-JMP-Befehl ein. Also verschiebt sich der Offset erneut. Damit sind alle Adressbezüge auf Daten in diesem Bereich falsch. In der Praxis ist dies höchst unbefriedigend, so daß die 8086-Assembler dem Programmierer die Möglichkeit einräumen, explizit den Typ des JMP-Befehls durch NEAR oder SHORT anzugeben. Damit lassen sich die Programme eindeutig gestalten und bereits bei der Entwicklung optimieren. Falls zu einer JMP SHORT-Anweisung das zugehörige Label außerhalb der Distanz von 127 Byte liegt, erzeugt der Assembler eine Fehlermeldung. Ich habe hier auf ein Beispiel verzichtet; vielleicht experimentieren Sie aber etwas mit dem Programm aus Listing 2.12 um die Wirkung des JMP-Befehls mit dem jeweiligen Präfix zu testen. Im Vorgriff auf den anschließend behandelten JMP FAR-Befehl enthält Bild 2.37 eine Aufstellung der drei Alternativen des JMP-Befehls.

Der JMP SHORT-Befehl besitzt folgende Syntax:

#### JMP SHORT Ziel

Mit Ziel wird hier eine Adresse im Codesegment bezeichnet, die maximal +127 und -128 Byte von der aktuellen Adresse entfernt sein darf. Der Abstand (Displacement) wird dabei relativ zur aktuellen Adresse angegeben. Beim JMP SHORT-Befehl ist deshalb eine Adressierung über Register oder indirekt über Speicherzellen nicht möglich. Vielmehr muß das Ziel als Konstante angegeben werden. Der Assembler berechnet dann das benötigte Displacement und setzt den Wert hinter dem Opcode ein.

#### Der JMP FAR-Befehl

Eine andere Situation herrscht vor, falls das Sprungziel über die Segmentgrenzen hinaus geht. Hier müssen sowohl die Segmentadresse im CS-Register, als auch der Offset im IP-Register neu gesetzt werden. Die Zieladresse läßt sich demnach nur mit 4 Byte darstellen. Der Befehl selbst umfaßt 5 Byte (1 Opcode und 4 Byte Adresse). Bei absoluten Adressangaben kann der Befehl zum Beispiel folgendermaßen dargestellt werden:

JMP 1000:3FFF

In der Regel wird man bei der Erstellung von Assemblerprogrammen aber das Sprungziel symbolisch darstellen. Dann wäre obiger Befehl als:

#### JMP WEITER

einzugeben. Hier besteht für den Assembler das Problem, daß er nicht erkennen kann, ob es sich um einen Sprung innerhalb des Segments oder über die Segmentgrenze hinweg handelt. Standardmäßig wird er obige Anweisung in einen JMP NEAR-Befehl umsetzen. Der Programmierer kann aber durch die Anweisung:

### JMP FAR WEITER

dem Assembler signalisieren, daß der Sprung über die Segmentgrenzen (FAR) geht und folglich eine 5 Byte lange Codefolge zu generieren ist.

Beim JMP FAR-Befehl läßt sich das Sprungziel nur direkt als 32-Bit-Konstante:

JMP FAR 1200:0033

JMP FAR Label

oder indirekt als Speicheradresse:

JMP FAR [33FF]

JMP FAR [BX+DI+3]

angeben. Im ersten Fall steht das Sprungziel als Konstante (z.B. EA 33 00 00 12) hinter dem JMP-Code. Im zweiten Fall liest der Prozessor die unter den Adressen [DS:33FF] oder [DS:BX+DI+3] abgespeicherte 32-Bit-Adresse und verzweigt zu dieser Programmstelle. Dabei ist zu beachten, daß das Sprungziel (z.B. 1200:0033) gemäß den INTEL-Konventionen:

Offset		Segment	
33	00	00	12

an der angegebenen Adresse gespeichert sein muß. Eine Adressierung über Register ist nicht möglich.

Nachfolgendes Beispiel (Listing 2.13) zeigt die Verwendung eines JMP FAR-Befehls mit indirekter Adressierung.

```

A 100
;=====
; File : RESET.ASM Born G. V 1.0
; DOS System Warmstart
; Der Reset-Vektor liegt ab Adresse
;      0000:0064
; in der Interrupt-Tabelle
;=====
;
JMP NEAR 112      ; an Programmbeginn
;
; Datenbereich mit dem Meldungstext
;
DB "System Reset", 0A, 0D, "$"
;
; Start:
MOV AH,09        ; DOS-Textausgabe
MOV DX,103       ; Textanfang
INT 21           ; DOS-Ausgabe
MOV AX,0000      ; ES auf 0000

```

```
MOV ES,AX      ; "  
ES:           ; indirekter Sprung  
JMP FAR [0064] ; Restart  
;  
; Programmende  
;  
  
N RESET.COM  
R CX  
20  
W  
Q
```

Listing 2.13: RESET.ASM

Die Aufgabe des Programmes ist es, unter DOS einen Warmstart durchzuführen. Die Adresse der Warmstartroutine findet sich in der Interrupt-Tabelle (INT 19) ab Adresse /1/:

0000:0064

Das Programm benutzt einen indirekten JMP FAR-Befehl zur entsprechenden Warmstartroutine. Hierfür setzt es den Inhalt des Registers ES auf Null und führt dann einen indirekten Sprung über die in [ES:0064] gespeicherte Adresse aus. Der Befehl:

ES:

definiert ein Segment-Override, damit der Zugriff auf die Adresse über ES:[] erfolgen kann. Das bedeutet, der Prozessor ermittelt erst die Adresse an der Speicherstelle ES:0064 (ES ist hier 0000) und liest den 32-Bit-Vektor in die Register CS:IP. Damit verzweigt er zu der angegebenen Stelle.

Zusammenfassend kann zum JMP-Befehl folgendes festgestellt werden:

- ◆ Es sind drei Variationen des unbedingten Sprungbefehls möglich, wobei diese eine unterschiedliche Anzahl von Codebytes (2 bis 5 Byte) besitzen.
- ◆ Bei der Anweisung JMP xxxx ist nicht immer klar, welche Variation des Befehls im Programm gemeint ist.
- ◆ Der Assembler generiert die jeweiligen Codes in Abhängigkeit vom Sprungziel.
- ◆ Letztlich ist der Programmierer dafür verantwortlich, daß die korrekte Variante des Befehls benutzt wird.
- ◆ Durch die Angabe des Prefix (NEAR, SHORT, FAR) läßt sich das Modell eindeutig festlegen.

Tabelle 2.36 enthält nochmals eine Zusammenstellung aller Möglichkeiten des JMP-Befehls mit den verschiedenen Variationen.

Sprungbefehl	Beispiel
JMP SHORT Disp8	JMP SHORT 1200 JMP SHORT Next
JMP NEAR Disp16	JMP NEAR 1300 JMP NEAR Weiter
JMP NEAR Mem16	JMP NEAR [1200] JMP NEAR [AX]
JMP NEAR Reg16	JMP NEAR AX JMP NEAR BP
JMP FAR Disp32	JMP FAR 1200:3FFF JMP FAR Label
JMP FAR Mem16	JMP FAR [1200]

Tabelle 2.36: Variationen des JMP-Befehls

### Die bedingten Sprungbefehle

Damit möchte ich das Thema unbedingte Sprünge verlassen und auf die bedingten Sprünge eingehen. Bei höheren Programmiersprachen gibt es Sprungbefehle, die nur in Abhängigkeit von einer vorher abzu prüfenden Bedingung auszuführen sind. Dort zum Beispiel die Verzweigung:

```
IF A > 10 THEN GOTO Exit;
```

erlaubt. Der Sprung wird nur ausgeführt, falls die Bedingung erfüllt ist. Die bisher besprochenen JMP-Befehle werden aber immer ausgeführt. Um nun auch bedingte Sprünge zu ermöglichen, haben die Entwickler der 80x86-Prozessoren einen ganzen Satz von Befehlen implementiert. Tabelle 2.37 gibt die 18 möglichen bedingten Sprungbefehle wieder.

Die CPU führt die Sprünge in Abhängigkeit von der getesteten Bedingung aus. Dabei ist allerdings festzuhalten, daß alle Sprungbefehle als SHORT implementiert sind, d.h. das Sprungziel darf maximal bis +127 und -128 Byte von der aktuellen Adresse entfernt liegen. Weiterhin ist zu beachten, daß viele dieser Sprungbefehle sich mit zwei verschiedenen mnemotechnischen Abkürzungen formulieren lassen (z.B. JA / JNBE), die dann aber durch den Assembler in einen Opcode umgesetzt werden! Bei der Disassemblierung mit DEBUG kann deshalb der Effekt auftreten, daß der ausgegebene Befehl nicht der ursprünglichen Anweisung entspricht (siehe Anhang). Nachfolgend werden die einzelnen Befehle besprochen.



Mnem.	Bedeutung	Test	Logik
JA/ JNBE	Jump if Above Jump if Not Below or Equal	$(CF \text{ AND } ZF)=0$	$X > 0$
JAE/ JNB	Jump if Above or Equal Jump if Not Below	$CF = 0$	$X \geq 0$
JB/ JNAE JC	Jump if Below Jump if Not Above or Equal Jump if Carry	$CF = 1$	$X < 0$
JBE JNA	Jump if Below or Equal Jump if Not Above	$(CF \text{ OR } ZF)=1$	$X \leq 0$
JCXZ JE/ JZ	Jump if CX ist Zero Jump if Equal Jump if Zero	$CX = 0$ $ZF = 1$	--- $A = B$ $X = 0$
JG/ JNLE	Jump if Greater Jump if Not Less nor Equal	$((SF \text{ XOR } OF) \text{ OR } ZF) = 0$	$X > Y$
JGE/ JNL	Jump if Greater or Equal Jump if Not Less	$(SF \text{ XOR } OF)=0$	$X \geq Y$
JL/ JNGE	Jump if Less Jump if Not Greater nor Equal	$(SF \text{ XOR } OF)=1$	$X < Y$
JLE/ JNG	Jump if Less or Equal Jump if Not Greater	$((SF \text{ XOR } OF) \text{ OR } ZF) = 1$	$X \leq Y$
JNC	Jump if No Carry	$CF = 0$	---
JNE/ JNZ	Jump if Not Equal Jump if Not Zero	$ZF = 0$	$X \neq Y$ $X \neq 0$
JNO	Jump if Not Overflow	$OF = 0$	---
JNP JPO	Jump if No Parity Jump on Parity Odd	$PF = 0$	---
JNS	Jump if No Sign	$SF = 0$	---
JO	Jump if Overflow	$OF = 1$	---
JP/ JPE	Jump if Parity Jump on Parity Even	$PF = 1$	---
JS	Jump on Sign	$SF = 1$	---

Tabelle 2.37: Bedingte Sprungbefehle

**Der Befehl JA /JNBE**

Der Sprungbefehl testet die Bedingung:

Jump if Above /  
Jump if Not Below or Equal

und verzweigt zum angegebenen Label, falls die obigen Bedingungen wahr sind. Dies ist offensichtlich der Fall, wenn der Wert größer Null ist (Jump if Above), oder falls der Wert nicht negativ oder Null ist (Jump if Not Below or Equal). Diese Bedingung läßt sich zwar durch:

$$X > 0$$

darstellen. Aber aus mir nicht ganz ersichtlichen Gründen hat INTEL zwei unterschiedliche Anweisungen (JA/JNBE) definiert. Diese werden vom Assembler aber mit dem gleichen Opcode (77 xx) dargestellt. Bei der Disassemblierung durch DEBUG erscheint nur der Befehl JA. Der Befehl besitzt die allgemeine Darstellung:

JA shortlabel  
JNBE shortlabel

Nachfolgend wird schematisch der Einsatz des Befehls gezeigt.

JA Gross

```
.
.
.
Gross: .
.
```

Die CPU prüft, ob das Carry- und das Zero-Flag den Wert 0 enthalten. In diesem Fall wird ein relativer Sprung zur Marke *Gross*: (Distanz maximal +127/-128 Byte) ausgeführt. Ist das Carry-Flag gesetzt, oder ist das Zero-Flag = 1, wird der Sprung nicht ausgeführt, sondern das Programm mit dem auf JA folgenden Befehl fortgesetzt.

### Der Befehl JAE /JNB

Der Sprungbefehl testet die Bedingung:

Jump if Above or Equal /  
Jump if Not Below

und verzweigt zum angegebenen Label, falls die obigen Bedingungen *wahr* sind. Dies ist der Fall, wenn der Wert größer oder gleich Null ist (Jump if Above or Equal), oder falls der Wert nicht negativ ist (Jump if Not Below). Diese Bedingung läßt sich durch:

$$X \geq 0$$

darstellen. Die CPU prüft, ob das Carry-Flag den Wert 0 enthält und führt dann einen relativen Sprung zur angegebenen Marke aus (Distanz maximal +127/-128 Byte). Ist das Carry-Flag gesetzt, unterbleibt der Sprung.

**Der Befehl JB /JNAE / JC**

Der Sprungbefehl testet die Bedingungen:

Jump Below /  
Jump if Not Above nor Equal /  
Jump if Carry

und verzweigt zum angegebenen Label, falls die obigen Bedingungen *wahr* sind. Dies ist der Fall, wenn der Wert kleiner Null ist oder falls das Carry-Flag gesetzt ist. Diese Bedingung läßt sich durch:

$$X < 0$$

darstellen. Die CPU prüft das Carry-Flag und führt den Sprung aus, falls das Flag gesetzt ist.

**Der Befehl JBE /JNA**

Der Sprungbefehl testet die Bedingungen:

Jump if Below or Equal /  
Jump if Not Above

und verzweigt zum angegebenen Label, falls die obigen Bedingungen *wahr* sind. Dies ist der Fall, wenn der Wert kleiner gleich Null ist. Diese Bedingung läßt sich durch:

$$X \leq 0$$

darstellen. Die CPU prüft das Carry-Flag und das Auxillary-Carry-Flag auf den Wert 1 ab und führt den Sprung aus, falls eines der Flag gesetzt ist. Bei  $CF = 0$  und  $ZF = 0$  erfolgt kein Sprung.

**Der Befehl JCXZ**

Der Sprungbefehl testet die Bedingung:

Jump if CX is Zero

und verzweigt zum angegebenen Label, falls die obige Bedingung zutrifft. Der Befehl prüft das Countregister CX auf den Wert Null. Die Bedingung läßt sich demnach zu:

$$CX = 0$$

formulieren. Mit dieser Abfrage lassen sich recht elegant Schleifen erzeugen:

```

A 100
;=====
; File : LOOP.ASM Born G. V 1.0
; gebe das Zeichen A auf dem
; Bildschirm N mal aus. N = Wert
; des CX-Registers
;=====
MOV CX,0003    ; Zähler = 3
MOV AH,02      ; Zeichenausgabe
MOV DL,41      ; Zeichen 'A'
; --> Schleifenanfang
INT 21         ; Write Zeichen
DEC CX         ; Zähler - 1
JCXZ 10E       ; Schleifenende?
JMP NEAR 107   ; LOOP
; Ende:
MOV AX,4C00    ; DOS-Exit
INT 21

R CX
50
N LOOP.COM
W
Q

```

*Listing 2.14: Schleifen mit dem JCXZ-Befehl*

Das kleine Programm gibt über die INT 21-Funktion 02H das Zeichen A auf dem Bildschirm aus. Das Zeichen ist im Register DL zu übergeben. Nun soll der Inhalt des Registers CX als Zähler dienen. Falls CX = 5 gesetzt wird, soll das Zeichen A 5 mal auf dem Bildschirm erscheinen. Der Befehl DEC CX subtrahiert jedesmal den Wert 1 vom Inhalt des Registers CX. Mit der Abfrage:

JCXZ xxx

prüft das Programm, ob der Zähler den Wert Null erreicht hat. In diesem Fall wird die Schleife verlassen. Andernfalls beginnt die Ausgabe des Zeichens per INT 21-Funktion 02H. Dann verzweigt die CPU über den unbedingten Sprung:

JMP NEAR XXX

zum Schleifenanfang. Da der Zähler vor jeder Ausgabe um den Wert 1 decrementiert und dann überprüft wird, bricht die Schleife bei CX = 0 ab.

### Der Befehl JE / JZ

Der Sprungbefehl testet die Bedingungen:

Jump if Equal /

Jump if Zero

und verzweigt zum angegebenen Label, falls die obigen Bedingungen wahr sind. Dies ist offensichtlich der Fall, wenn der Wert einer vorausgehenden Operation das Ergebnis 0 geliefert hat. Die CPU prüft das Zero-Flag und führt den Sprung aus, falls das Flag gesetzt (1) ist. Das folgende kleine Programm zeigt den Einsatz des Befehls mit den zwei verschiedenen Abkürzungen.

```
A 100
;=====
; File: FRAGE.ASM (c) Born G. V 1.0
; Abfrage der Tastatur auf das
; Zeichen J. Wird dieses Zeichen
; erkannt, terminiert das Programm
; mit der Meldung: OK
; Bei 5 Fehlversuchen terminiert
; das Programm mit einer Fehlermeld.
;=====
MOV AX,0900      ; Eröffnungsmeldung
MOV DX,12E       ; Ptr = Text
INT 21           ; Ausgabe
; init Variable
MOV CX,0         ; init Counter
; LOOP:      Eingabeschleife
MOV AH,01        ; Read Keyboard & Echo
INT 21           ; "
; Teste Zeichen auf J
CMP AL,4A        ; Zeichen = J?
JE 122          ; JMP -> OK
INC CX           ; Count + 1
MOV AX,05        ; Count = 5 ?
SUB AX,CX        ; "
JZ 11D           ; JMP -> Fehler
JMP SHORT 10B    ; JMP -> Loop
; Fehlerausgang
MOV DX,153       ; PTR = Fehlertext
JMP SHORT 125    ; Ausgabe
; OK
MOV DX,17D       ; PTR = Text "OK"
; Ausgabe
MOV AH,09        ; DOS Stringausgabe
INT 21
; Exit
MOV AX,4C00      ; terminate
INT 21
;
;
; Datenbereich
;
;
DB "Demo, bitte das Zeichen J eingeben",0D,0A,"$"
DB "Falscheingabe, Abbruch nach 5 Versuchen",0D,0A,"$"
DB 0D,0A,"OK - > Ende der DEMO",0D,0A,"$"
;
;
; Ende -> erzeuge COM-File in DEBUG
;
;
```

```
R CX
100
N DEMO.COM
W
Q
```

*Listing 2.15: Einsatz des JE/JZ-Befehls*

Die Routine dient zur Abfrage der Tastatur auf das Zeichen 'J'. Ist dies der Fall, wird die Meldung:

OK

ausgegeben. Falls die Abfrage 5 mal durchgeführt wurde, bricht das Programm mit einer Fehlermeldung ab. Die Tastatur läßt sich durch die INT 21-Funktion 01 abfragen. Das Ergebnis wird im AL-Register zurückgegeben /1/. Der Befehl:

CMP AL,4A

prüft nun, ob der Inhalt des Registers AL mit der angegebenen Konstanten (hier das Zeichen J) übereinstimmt. In diesem Fall wird das Zero-Flag (Equal) gesetzt. Mit der SUB-Anweisung wird der Inhalt des Registers CX vom Inhalt des Registers AX subtrahiert. Da AX mit der Konstanten 5 geladen wird, ergibt sich bei CX = 5 als Ergebnis der Subtraktion der Wert 0. CX dient aber als Zähler und wird bei jedem Durchlauf mit dem Befehl INC CX erhöht, so daß das Programm bei CX = 5 abbricht.

#### Der Befehl JG /JNLE

Der Sprungbefehl testet die Bedingungen:

Jump Greather /  
Jump if Not Less nor Equal

und verzweigt zum angegebenen Label, falls die obigen Bedingungen *wahr* sind. Hierzu prüft die CPU, ob das Sign-Flag und das Overflow-Flag den gleichen Wert (0 oder 1) haben und ob das Zero-Flag auf Null gesetzt ist. In diesem Fall wird der Sprung ausgeführt.

#### Der Befehl JGE /JNL

Der Sprungbefehl testet die Bedingungen:

Jump Greather or Equal /  
Jump if Not Less

und verzweigt zum angegebenen Label, falls die obigen Bedingungen wahr sind. Hierzu prüft die CPU, ob das Sign-Flag und das Overflow-Flag den gleichen Wert (0 oder 1) haben.

#### Der Befehl JL /JNGE

Der Sprungbefehl testet die Bedingungen:

Jump Less /  
Jump if Not Greather nor Equal

und verzweigt zum angegebenen Label, falls die obigen Bedingungen *wahr* sind. Hierzu prüft die CPU, ob das Sign-Flag und das Overflow-Flag ungleiche Werte (0 oder 1) haben. Nur in diesem Fall wird der Sprung ausgeführt.

#### Der Befehl JLE /JNG

Der Sprungbefehl testet die Bedingungen:

Jump if Less or Equal /  
Jump if Not Greather

und verzweigt zum angegebenen Label, falls die obigen Bedingungen *wahr* sind. Die CPU prüft, ob das Sign-Flag und das Overflow-Flag ungleiche Werte (0 oder 1) haben, oder ob das Zero-Flag auf Eins gesetzt ist. In diesen Fällen wird der Sprung ausgeführt.

#### Der Befehl JNC

Der Sprungbefehl testet die Bedingung:

Jump if Not Carry

und verzweigt zum angegebenen Label, falls das Carry-Flag nicht gesetzt (0) ist.

#### Der Befehl JNE /JNZ

Der Sprungbefehl testet die Bedingungen:

Jump if Not Equal /  
Jump if Not Zero

und verzweigt zum angegebenen Label, falls die obigen Bedingungen wahr sind. Es wird nur geprüft, ob das Zero-Flag gleich Null ist. In diesen Fällen wird der Sprung ausgeführt. Das nachfolgende Beispiel aus Listing 2.16 nutzt diesen Befehl um einen Text zeichenweise über die Funktion 02H des INT 21 auszugeben. Als Besonderheit sei auf die Lage des Textes hingewiesen:

**Anmerkung:** Wird beim Aufruf eines Programmes hinter dem Programmnamen ein Text (Parameterstring) angegeben, legt DOS diesen bis zu 127 Byte langen String in einem Puffer im Programm-Segment-Prefix (PSP) des Programmes ab. Dieser PSP ist ein 256 Byte langer Datenbereich, der vor jedem geladenen Programm in den Adressen CS:0000 bis CS:00FF von DOS angelegt wird. In diesem Bereich finden sich Informationen für DOS über das Programm (z.B.: Rückkehradresse beim Programmende, Rückkehradresse beim Fehlerabbruch, Text des Parameterstrings, Filehandles, etc.). Die genaue Belegung des weitgehend undokumentierten PSP-Bereichs ist in /1/ beschrieben.

Der PSP-Bereich ist die Ursache, daß ein Programm erst ab CS:100 beginnen darf. Deshalb wird bei DEBUG immer die Anweisung "A 100" am Programmanfang eingegeben! Das Programm, nennen wir es DEMO.COM werde zum Beispiel mit folgen der Eingabe aufgerufen:

DEMO Hallo

Dann findet sich ab Adresse CS:80 die Struktur gemäß Bild 2.38 im PSP:

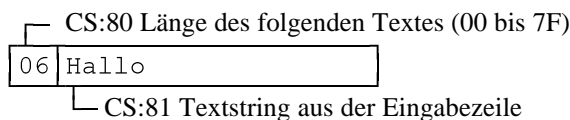


Bild 2.38: Aufbau des Transferpuffers im PSP-Bereich

Auf der Adresse CS:80 steht immer die Zahl der nachfolgenden Zeichen. Der Wert 0 signalisiert, daß keine Parameter vorhanden sind. Ein Wert ungleich 0 markiert einen nachfolgenden String. Das erste Zeichen an der Adresse CS:81 ist immer ein Leerzeichen (es ist das Leerzeichen, welches den Programmnamen vom Parameterstring trennt). Daran schließen sich die in der Kommandozeile eingegebenen weiteren Zeichen an. Der Text wird mit 0D abgeschlossen, wobei dieses nicht mehr zum Text gehört und auch im Längenbyte nicht berücksichtigt wird. Dieser Sachverhalt läßt sich leicht mit DEBUG überprüfen.

Nachdem die Aufgabe halbwegs klar erscheint, beginnt die Umsetzung in ein (hoffentlich) lauffähiges Programm. Nachfolgend wird einer der möglichen Ansätze vorgestellt.



Programmbeispiel

Bei der Erstellung des Assemblerprogramms wird vielfach die Frage gestellt, welche Register verwendet werden sollen. Einmal benötigen wir einen Längenzähler für die Zahl der auszugebenden Bytes. Dafür kann man ein 8-Bit-Register (AH, AL, BH, BL, CH, CL, DH, DL) verwenden. Beachten Sie aber, daß vielleicht einige Register später noch gebraucht werden. AX wird häufig bei INT 21-Aufrufen belegt, wodurch ich auf eine Verwendung verzichten würde. DL ist für Textausgabe per INT 21 reserviert, fällt also auch weg. BX eignet sich sehr gut als Zeiger, wodurch nur noch ein 8-Bit-Register von CX bleibt. Als Zeiger auf die Stringadresse dürfen grundsätzlich die Register BX, DI, SI und BP verwendet werden. Da BP das Stacksegment (SS) für Adresszugriffe benutzt, entfällt diese Möglichkeit. DI und SI wären eine Lösung, ich habe mich aber hier für BX als Zeiger auf den String entschieden.

Da bei COM-Programmen die Segmentregister durch DOS beim Programmstart auf CS = DS = SS = ES gesetzt werden, funktioniert der Befehl:

```
MOV CL,[80]
```

Bei EXE-Programmen geht dies schief, da der Kommandostring bei CS:80 steht, der MOV-Befehl aber ein Datum bei DS:80 liest! Abhilfe schafft hier ist notfalls ein Segment-Override mit CS.

Zur Ausgabe eines Zeichens bietet DOS die Unterprogrammfunktion AH = 02 des INT 21. Diese Funktion erwartet in DL das auszugebende Zeichen (siehe Anhang und /1/).

```
A 100
;-----
; Demoprogramm zur Textausgabe  (Born G.)
; File:   DEMO.ASM  V 1.0
; Aufruf: DEMO Parameterstring
;         Der Parameterstring ist optional und
;         wird vom Programm auf dem Bildschirm
;         ausgegeben. Der Aufruf:
;
;         DEMO Hallo
;
;         gibt den Text:
;
;         Hallo
;
;         aus. Der Text steht ab CS:81, wobei
;         ab CS:80 die Länge des Strings steht.
;         Das Programm ist mit:
;
;         DEBUG < DEMO.ASM > DEMO.LST
;
;         zu übersetzen.
;-----
; Start:   Beginn Programmcode ab CS:100 !!!
MOV      CL,[80]      ; lese Länge in DS:80
```

```

AND     CL,FF      ; Zeichenzahl = 0 ?
JZ      117        ; Ja -> JMP EXIT ### Adresse
;
; Text ist vorhanden, ausgeben
;
MOV     BX,81      ; Zeiger auf 1. Zeichen
; Loop: Beginn der Ausgabeschleife !!!!
MOV     AH,02      ; DOS-Code Zeichenausgabe
MOV     DL,[BX]    ; Zeichen in DL laden
INT     21         ; CALL DOS-Zeichenausgabe
INC     BX         ; Zeiger auf nächstes Zeichen
DEC     CL         ; Zeichenzahl - 1
JNZ     10C        ; <> 0 -> JMP Loop ### Adresse
;
; Text ist ausgegeben, Programm beenden mit INT 21
; Register: AH = 4C, AL = 00 (Fehlercode)
;
; Exit:
MOV     AX,4C00    ; DOS-Exitcode
INT     21         ; CALL-DOS-Exit
; Programmende, Leerzeile folgt und dann die An-
; weisungen für die Speicherung nicht vergessen !

N DEMO.COM
R CX
200
W
Q

```

Listing 2.16: Das Programm DEMO.ASM

Listing 2.16 zeigt das fertige Programm. Mit dem Befehl:

AND CL,FF

läßt sich prüfen, ob CL = 0 ist. Der Befehl verändert in diesem speziellen Fall nur die Flags und nicht den Inhalt von CL. Dies ist beim AND-Befehl normalerweise nicht der Fall. Besser wäre hier die Verwendung des CMP-Befehls gewesen, da er für Vergleiche vorgesehen ist. Ich wollte aber zeigen, daß Vergleiche eventuell auch mit anderen Befehlen durchführbar sind. Mit der Anweisung JZ 11A wird zum Programmende verzweigt, falls keine Zeichen vorliegen. Hier tritt bei der Programmerstellung mit DEBUG wieder das Problem mit der absoluten Adresse auf. Im ersten Schritt wird die Konstante 100 als Dummy-Sprungziel eingegeben. Nach einer fehlerfreien Übersetzung läßt sich aus dem Listing die gesuchte Adresse entnehmen und im Quellprogramm eintragen. Beachten Sie aber, daß ein JMP SHORT nur über 127 Byte geht. Die Zieladresse darf daher nie weiter als diese 127 Byte entfernt liegen, sonst gibt es eine Fehlermeldung beim Übersetzen.

### Der Befehl JNO

Der Sprungbefehl testet die Bedingung:

### Jump Not Overflow

und verzweigt zum angegebenen Label, falls das Overflow-Flag den Wert 0 besitzt.

### Der Befehl JNP / JPO

Der Sprungbefehl testet die Bedingungen:

Jump if No Parity /  
Jump if Parity Odd

und verzweigt zum angegebenen Label. Der Befehl prüft das Parity-Flag auf den Wert 0 ab und führt in diesem Fall den Sprung aus. Das Parity-Flag wird gesetzt, falls die Zahl der gesetzten Bits in dem Datenbyte gerade (even) ist. Bei ungerader Anzahl von Einsbits ist das Parity-Flag gelöscht.

### Der Befehl JNS

Der Sprungbefehl testet die Bedingung:

Jump No Sign

und verzweigt zum angegebenen Label, falls die obige Bedingung erfüllt ist. Hierzu muß das Sign-Flag den Wert 0 besitzen. Dieses Flag gibt an, ob das oberste Bit einer Zahl gesetzt ist. Bei vorzeichenbehafteten Zahlen entspricht dies dann einem negativen Wert.

### Der Befehl JO

Der Sprungbefehl prüft die Bedingung:

Jump if Overflow

Das Overflow Flag wird gesetzt, falls eine arithmetische Operation (Addition, Multiplikation) zu einem Überlauf führt. In diesem Fall kann das Ergebnis nicht mehr in den verwendeten Registern dargestellt werden. Mit Hilfe des JO-Befehls läßt sich dann zu einer Fehlerroutine springen.

### Der Befehl JP / JPE

Der Sprungbefehl testet die Bedingungen:

Jump on Parity /

### Jump if Parity Even

und verzweigt zum angegebenen Label, falls das Parity-Flag den Wert 1 besitzt. Das Parity-Flag wird gesetzt, falls die Zahl der Bits mit dem Wert 1 in dem Datenbyte gerade (even) ist. Bei ungerader Anzahl von Einsbits ist das Parity-Flag gelöscht.

### Der Befehl JS

Der Sprungbefehl testet die Bedingung:

#### Jump if Sign

und verzweigt zum angegebenen Label, falls die obige Bedingung erfüllt ist. Hierzu prüft die CPU, ob das Sign-Flag den Wert 1 besitzt. Dies ist bei negativen Zahlen der Fall.

Die Konditionen zur Ausführung der bedingten Sprungbefehle sind in Tabelle 2.35 aufgeführt. Die einzelnen Flags werden durch verschiedene Operationen auf Daten (ADD, AND, CMP, etc.) gesetzt. Welcher Befehl welche Flags setzt wird bei der Beschreibung der einzelnen Befehle diskutiert. Mit:

#### AND AX,AX

läßt sich zum Beispiel prüfen, ob der Inhalt des Registers AX den Wert Null besitzt. Weitere Beispiele finden sich in den Listings der folgenden Kapitel.

Die Sprungbefehle verändern die Flags nicht. Damit bleibt nur noch die Frage, wie sich bedingte Sprünge über Distanzen von mehr als 127 Byte ausführen lassen. Die CPU bietet hierzu keine Befehle. Mit einem kleinen Trick lassen sich dennoch bedingte Sprünge über beliebige Distanzen ausführen. Hierzu werden einfach bedingte und unbedingte Sprünge nach folgender Methode kombiniert:

```
.  
.  
JNC Weiter ; Fortsetzung  
JMP Carry ; Carry gesetzt  
Weiter: .  
.  
Carry: .
```

Vor den eigentlichen Sprung wird eine bedingte Sprunganweisung mit negierter Abfrage gesetzt. Ist die ursprüngliche Bedingung nicht wahr, wird die folgende JMP-Anweisung übergangen. In unserem Beispiel war es das Ziel, bei einem gesetzten Carry-Flag einen Sprung über größere Distanzen als 127 Byte auszuführen. Also erfolgt die Abfrage auf No Carry in der vorhergehenden bedingten Sprunganweisung.

Mit dieser Technik lassen sich sowohl bedingte NEAR- als auch bedingte FAR-Sprünge ausführen.

### 2.10.2 Die CALL-Befehle

Bei der Programmentwicklung besteht der Wunsch, die Struktur durch Modularisierung übersichtlich zu halten. In Hochsprachen werden häufig Prozeduren oder Unterprogramme für diesen Zweck eingesetzt. Auch der 8086-Befehlsvorrat bietet den CALL-Befehl, um Unterprogramme aufzurufen. Dieser Befehl besitzt das allgemeine Format:

CALL {Len} Ziel

Mit Ziel wird die Anfangsadresse des Unterprogrammes angegeben. Die CPU unterbricht dann das Hauptprogramm an der aktuellen Adresse um an der neuen Adresse aufzusetzen. Im Gegensatz zum JMP-Befehl merkt sich der Prozessor aber die Adresse, an der der CALL-Befehl gelesen wurde auf dem Stack. Damit besteht die Möglichkeit nach Beendigung des Unterprogrammes an die Unterbrechungsstelle zurückzukehren. Ähnlich wie beim JMP-Befehl gibt es beim CALL-Aufruf verschiedene Formen, die durch optionale Schlüsselworte im Feld *Len* selektiert werden.

#### Der CALL-NEAR-Befehl

Dieser Befehl erlaubt den Aufruf von Unterprogrammen innerhalb eines 64-Kbyte-Programmsegmentes. Der Befehl besitzt die allgemeine Form:

CALL NEAR Ziel

und wird vom Assembler mit 3 Byte kodiert. Das Schlüsselwort NEAR signalisiert dabei dem Assembler, daß es sich um einen Aufruf innerhalb des Segmentes handelt. In diesem Fall wird nur der Inhalt des Instruktionpointers auf dem Stack gesichert und dann wird aus *Ziel* die neue Startadresse gelesen (Bild 2.39).

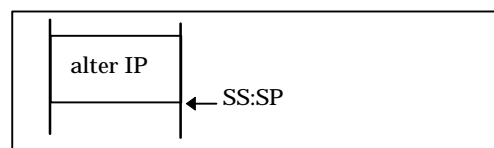


Bild 2.39: Stackzustand beim CALL NEAR-Befehl.

Die CPU wendet bei Unterprogrammaufrufen mit direkter Adreßangabe ebenfalls die relative Adressierung an, so daß Distanzen von +/- 32 KByte überwunden werden können. Der Vorteil der relativen Adressierung liegt, wie bereits beim JMP-Befehl erwähnt, in der Möglichkeit zur Erzeugung relokativen Codes, d.h. die Programme

können innerhalb der 64-KByte frei verschoben werden und bleiben trotzdem lauffähig. Der CALL-NEAR-Befehl erlaubt allerdings mehrere Adressierungsarten:

```
CALL NEAR Mult           ; "
CALL NEAR [3000]         ; indirekt über den
                          ; Speicher
CALL NEAR [BP+SI+2]      ; "
CALL NEAR AX             ; Indirekt über Register
```

Am einfachsten ist die direkte Adressierung, bei der die Zieladresse direkt als Konstante oder Label angegeben wird. Damit muß die Adresse bereits bei der Programmerstellung bekannt sein. Alternativ erlaubt der CALL NEAR-Befehl eine indirekte Adressierung bei der das Sprungziel zum Beispiel in einem der 16-Bit-Universalregister übergeben wird. Die Sprungadresse darf aber auch in einer Speicherzelle abgelegt werden. Diese Zelle läßt sich dann über verschiedene Register indirekt adressieren (z.B. CALL NEAR [BX+3000]). Die Zieladresse wird dabei standardmäßig aus dem Datensegment gelesen. Nur bei Verwendung des BP-Registers innerhalb der indirekten Adresse (z.B. [BP+DI+3]) liegt die Speicherzelle mit der Zieladresse im Stacksegment. Durch einen Segment-Override-Befehl läßt sich diese Einstellung allerdings überschreiben. Alle CALL-Aufrufe, bei denen die Adresse indirekt über ein Register oder eine Speicherzelle ermittelt wird, benutzen aber eine absolute Adressierung für das Sprungziel. Die Sequenz:

```
MOV AX, 2000
CALL NEAR AX
```

führt damit einen Sprung zur Adresse CS:2000 aus.

Enthält ein CALL-Aufruf kein Schlüsselwort (NEAR oder FAR), wird immer ein CALL NEAR generiert. Manche Assembler verlangen bei der indirekten Adressierung die Schlüsselworte:

WORD PTR

um den Befehl zu akzeptieren. Ein CALL-Aufruf sieht dann folgendermaßen aus:

```
CALL WORD PTR [BX+10]
```

Die Zieladresse findet sich in der durch DS:BX+10 adressierten Speicherstelle. Um die Auswirkungen des CALL-Befehls zu studieren habe ich das Programm aus Listing 2.17 entwickelt.

```
A 100
;=====
; File: CALL1.ASM
; Funktion: Demonstration des CALL NEAR Befehls
;           Ein Unterprogramm zur Ausgabe von
```

```

; Zeichen wird aufgerufen.
;=====
; Assembliere ab Adresse 100H
; Beginn des Hauptprogrammes
;
; Start:
; Aufruf der Ausgaberroutine per direktem CALL
;
;     MOV DL,31      ; 1 laden
;     CALL NEAR 200  ; Ausgaberroutine rufen
;
; Aufruf der Ausgaberroutine per indirektem CALL
; über den Inhalt des Registers AX
;
;     MOV DL,32      ; 2 laden
;     MOV AX,200     ; Adresse Unterprogramm
;     CALL NEAR AX   ; Ausgaberroutine rufen
;
; Aufruf der Ausgaberroutine per indirektem CALL
; über den Inhalt der Speicherzelle DS:150
; In einer COM-Datei ist DS = CS = SS = ES !!!
;
;     MOV AX,200     ; init Speicherstelle
;     MOV [150],AX   ; mit dem Sprungziel
;     MOV DL,33      ; 3 laden
;     CALL NEAR [150] ; Ausgaberroutine rufen
;
; Aufruf der Ausgaberroutine per indirektem CALL
; über den Inhalt der durch BX adressierten Zelle
;
;     MOV DL,34      ; 4 laden
;     MOV AX,200     ; Adresse Unterprogramm
;     MOV [150],AX   ; initialisieren
;     MOV BX,150     ; lese Zeiger
;     CALL NEAR [BX] ; Ausgaberroutine rufen
;
; Rückkehr zu MS-DOS
;
;     MOV AX,4C00    ; DOS-Code "Exit"
;     INT 21         ; Terminiere Programm
; hier muß eine Leerzeile kommen
A 200
;=====
; Unterprogramm Output
; Die Routine gibt das in DL übergebene ASCII-Zeichen
; auf dem Bildschirm aus und hängt die Nachricht:
; " . Aufruf der Routine<CR/LF>"
; an.
;=====
; Output:
;     MOV AH,02      ; DOS-Code "Write Char"
;     INT 21         ; ASCII-Zeichen ausgeben
;     MOV DX,20C     ; lade Stringadresse
;     MOV AH,09      ; DOS-Code "Write String"
;     INT 21         ; String ausgeben
;     RET           ; Ende Unterprogramm
;=====
; Datenbereich mit dem Textstring

```

```
;/=====
DB ". Aufruf der Routine",0D,0A,"$"
;
; Steueranweisungen für DEBUG

N DEMO1.COM
R CX
250
W
Q
```

*Listing 2.17: Demonstration des CALL NEAR-Befehls.*

Dieses enthält eine Unterroutine zur Ausgabe eines Textes. Dabei wird das im Register DL befindliche ASCII-Zeichen auf dem Bildschirm ausgegeben und mit dem Text:

Aufruf der Routine

ergänzt. Das Hauptprogramm enthält verschiedene Variationen des CALL-NEAR-Befehls zum Aufruf der Ausgaberroutine. Vielleicht bearbeiten Sie dieses Programm mit DEBUG und verfolgen den Ablauf mit der Trace-Anweisung.

### Der CALL FAR-Befehl

Soll ein Unterprogrammaufruf über die Segmentgrenzen hinaus erfolgen, bietet der 8086-Prozessor den CALL FAR-Befehl. Dieser besitzt folgendes Format:

CALL FAR Ziel

Das Schlüsselwort FAR muß dabei immer angegeben werden, während die Zieladresse entweder direkt oder indirekt über eine Speicherzelle spezifiziert wird. Eine Adressierung über Register in dagegen nicht möglich. Nachfolgend sind einige Befehle aufgeführt.

```
CALL FAR 3FFF:0100    ; direkt absolute Adr.
CALL FAR bios         ; direkt über Labels
CALL FAR [3000]       ; indirekt über MEM
CALL FAR [BX+SI+20]   ;          "
```

Der Befehl benötigt einen 32-Bit-Adreßvektor als Ziel und wird bei der direkten Adressierung mit 5 Byte kodiert. Bei der indirekten Adressierung ist der 32-Bit-Vektor im Datensegment abzulegen. Nur bei Verwendung des BP-Registers bezieht sich die Adreßangabe auf das Stacksegment. Beim Aufruf sichert der CALL-Befehl den Inhalt des CS- und IP-Registers gemäß Bild 2.40 auf dem Stack.



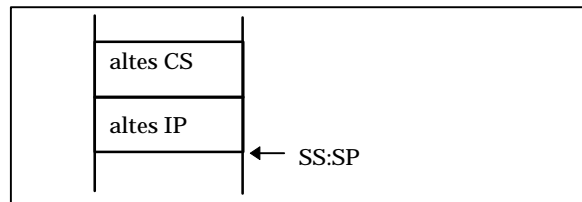


Bild 2.40: Stack beim Aufruf des CALL FAR-Befehls

Einige Assembler erwarten bei der indirekten Adressierung die Schlüsselworte:

DWORD PTR

wodurch der Befehl zum Beispiel folgendes Format annimmt:

CALL FAR DWORD PTR [3000]

Auf ein Programm zur Anwendung des CALL FAR-Befehls wird an dieser Stelle verzichtet, da die Aufrufe im wesentlichen der Struktur in Listing 2.18 entsprechen. Lediglich die Adressierung über Register ist nicht möglich.

### Der RET-Befehl

In Listing 2.17 wurde bereits ein neuer 8086-Befehl zum Beenden des Unterprogrammes eingeführt. Sobald im Programm ein RET auftaucht, liest die CPU die Rückkehradresse vom Stack und setzt das unterbrochene Hauptprogramm fort. Für die korrekte Beendigung eines Unterprogrammes müssen natürlich einige Bedingungen beachtet werden:

- ◆ Alle vom Unterprogramm auf dem Stack gesicherten Parameter sind vorher zu entfernen.
- ◆ In Abhängigkeit vom CALL-Aufruf ist ein entsprechender RET-Befehl auszuführen.

Für beide Bedingungen ist der Programmierer verantwortlich. Die Nichtbeachtung dieser einfachen Regeln ist häufig der Grund für schwer zu lokalisierende Programmabstürze.

Da es zwei verschiedene CALL-Befehle (NEAR, FAR) gibt, existieren auch die entsprechenden Befehle:

RET ; für CALL NEAR  
RETF ; für CALL FAR

Bei einem Aufruf mit einem CALL NEAR muß die Routine auch mit einem normalen RET beendet werden. Die Anweisung CALL FAR legt dagegen eine 4 Byte lange Rücksprungadresse auf dem Stack ab. Mit einem RET werden aber nur 2 Byte entfernt. Deshalb gibt es den RETF-Befehl, der die 4 Byte vom Stack in die Register CS:IP zurückliest und einen Rücksprung über Segmentgrenzen erlaubt. Der wechselseitige Aufruf eines Unterprogrammes über CALL FAR und CALL NEAR ist damit nicht möglich. Einige Assembler kennen nur den Befehl RET und setzen in Abhängigkeit vom verwendeten Modell den Code für ein RET oder RETF ein. Dies ist allerdings eine recht tückische Sache, insbesondere wenn Anfänger die Listings abtippen. Die Fehler sind häufig nur sehr schwer zu lokalisieren.

Im Zusammenhang mit dem RET-Befehl möchte ich noch auf eine weitere Eigenschaft hinweisen. Oft werden dem Unterprogramm Parameter vom rufenden Programm übergeben. Dies kann wie in Listing 2.18 über die Register oder über Zeiger erfolgen. Oft nutzen Softwareentwickler aber die Parameterübergabe per Stack. Nun sollte das Unterprogramm diese Parameter vor der Rückkehr in das rufende Programm entfernen. Gehen wir einmal von folgender Aufrufsequenz aus:

```
MOV AX,100      ; Parameter 1
PUSH AX
CALL NEAR TEST
```

Für das Unterprogramm liegt dann ein Stackzustand gemäß Bild 2.41 vor.

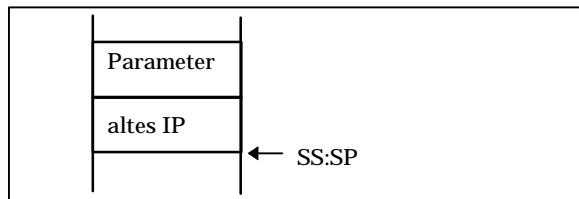


Bild 2.41: Parameterübergabe per Stack

Nun tritt häufig der Fall auf, daß das Unterprogramm die Parameter lediglich liest. Dann müssen die Werte vor Ausführung der RET-Anweisung explizit vom Stack entfernt werden. Dies ist mit folgender Sequenz:

```
POP AX      ; Rückkehradresse
POP BX      ; Parameter
PUSH AX     ; restore Adresse
RET
```

möglich. Mich stört allerdings die Zahl der Befehle und weiterhin wird der Inhalt einiger Register zerstört. Die Entwickler haben deshalb dem RET-Befehl die Möglichkeit gegeben, mehrere Worte vom Stack zu entfernen. Der Aufruf:

## RET 2

erledigt die obige Aufgabe eleganter als die Sequenz aus POP- und PUSH-Befehlen. Zuerst wird die Rückkehradresse gelesen und dann der Stackpointer um 4 (2 Worte) erhöht. Damit ist der Parameter vom Stack entfernt.

### 2.10.3 Der INT-Befehl

Eine weitere Möglichkeit zum Aufruf von Unterprogrammen bieten die Interrupt-Befehle. In einigen Beispielen wurde bereits der INT 21-Befehl zum Aufruf von DOS-Routinen benutzt. Ein INT löst eine Programmunterbrechung aus. Die CPU sichert dann den Inhalt des Flagregisters und die Rücksprungadresse gemäß Bild 2.42 auf dem Stack.

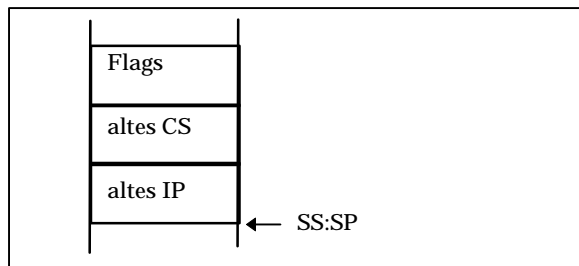


Bild 2.42: Stackzustand beim INT-Befehl

Dann liest der Prozessor einen 4-Byte-Vektor mit der Zieladresse der Interruptroutine aus einer Tabelle ein und verzweigt zu dieser Adresse. Diese Interrupt-Vektor-Tabelle liegt im unteren Adreßbereich von 0000:0000 bis 0000:03FFH und besitzt eine Länge von 1 KByte. Die 8086-CPU's können 256 verschiedene Interrupts unterscheiden. Jedem dieser Interrupts ist in dieser Tabelle ein 4-Byte-Vektor zugeordnet. Das Betriebssystem oder ein Programm kann in dieser Tabelle Vektoren auf eigene Programmteile eintragen, die dann bei der Aktivierung des betreffenden Interrupts aufgerufen werden. Das DOS-Betriebssystem benutzt zum Beispiel den INT 21-Vektor um Anwenderprogrammen Systemroutinen zur Verfügung zu stellen. Die Anwenderprogramme benötigen dann nur noch die Information, welche Parameter zu übergeben sind, während die Adresse der betreffenden Routine unbekannt bleibt. Der INT-Befehl besitzt zusätzlich noch den Vorteil, daß er mit maximal 2 Byte kodiert wird. Der Aufruf:

```
MOV AH,4C    ; DOS exit
MOV AL,00    ; ERRORLEVEL
INT 21        ; terminate
```

beendet z.B. unter MS-DOS ein Programm und gibt die Kontrolle an das Betriebssystem zurück. Die Sequenz wurde bereits mehrfach in Beispielprogrammen benutzt. Im Register AH ist der INT 21-Funktion ein Befehlscode (hier 4CH) zu übergeben. AL dient bei der Funktion 4CH zur Übergabe eines Exitcodes, der sich in DOS durch die ERRORLEVEL-Funktion abfragen läßt. Nähere Hinweise zu den DOS-INT 21-Aufrufen finden sich im Anhang und in /1/.

Mit der INT-Technik lassen sich Unterprogramme (bezeichnet als Interrupt-Service-Routinen) aufrufen, ohne daß das Programm deren Adresse kennen muß. Ein Interrupt kann softwaremäßig durch einen INT xx-Befehl oder per Hardware über den INT-Eingang des Prozessors aktiviert werden. Bei der Hardwareunterbrechung sorgt ein eigener Baustein (Interrupt-Controller) für die Generierung des INT xx-Befehls. In einem PC werden zum Beispiel die Tastatureingaben, die Uhrzeit, etc. per Hardware-interrupt verarbeitet.

Zwei Interrupts nehmen eine Sonderstellung ein:

INT0  
INT3

Der INTO-Befehl wird nur ausgeführt, falls das Overflow-Flag gesetzt ist. Dies kann bei der Anwendung arithmetischer Befehle nützlich sein. Der INT3 wird durch ein Opcodebyte kodiert. Deshalb benutzen viele Debugger diesen Befehl zum Setzen von Unterbrechungspunkten. Sobald DEBUG in einem Programm den INT 3 findet, unterbricht er den Programmablauf und meldet sich mit dem Promptzeichen (-).

### Der IRET-Befehl

Eine Interruptroutine darf nicht mit einem einfachen RETF-Befehl beendet werden. In diesem Fall wird zwar die Rückkehradresse korrekt zurückgelesen, der Inhalt des Flagregisters verbleibt aber auf dem Stack. Bei mehrfachem Aufruf kommt es dann zu einem Stacküberlauf. Das Problem läßt sich zwar mit der Anweisung:

RETF 2

lösen, aber die 8086-Anweisung:

IRET

sorgt nicht nur für die korrekte Restaurierung des Stacks, sondern stellt auch den Inhalt des Flagregisters auf den Zustand vor Aufruf des Interrupts wieder her. Dies ist in vielen Fällen recht hilfreich, da ja der Zustand der Flags in der Interruptroutine verändert werden kann. Das Programm aus Listing 2.18 demonstriert den Umgang mit verschiedenen BIOS- und DOS-Interrupts. Speichern Sie die Anweisungen in einer Textdatei mit dem Namen INT.ASM und übersetzen diese mit DEBUG:

## DEBUG &lt; INT.ASM

Auf dem Bildschirm erscheint das Listing und in INT1.COM steht später der ausführbare Code. Dieser läßt sich mit der Eingabe:

INT1

starten. Dann sollte auf dem Bildschirm ein inverses Fenster mit dem Text erscheinen.

```
A 100
;=====
; File: INT1.ASM
; Funktion: Demonstration des INT Befehls für
;           BIOS- und DOS-Zugriffe.
;=====
;   Assembliere ab Adresse 100H
;
; Start:
; Up Scroll des Bildschirms (clear) per INT 10 Funktion
; AH 07H, AL = Zeilenzahl -> 0 = clear window
; CH = Eckzeile links oben, CL = Eckspalte
; DH = Eckzeile unten rechts, DL = Eckspalte
;
;           MOV  AX,0600      ; up scroll, clear
;                               ; window
;           MOV  BH,07        ; Attribut normal
;           MOV  CX,0000      ; linke obere Ecke
;           MOV  DX,1850      ; rechte untere Ecke
;           INT  10           ; BIOS Routine rufen
;
; Down Scroll eines Fensters per INT 10 Funktion
; AH 07H, AL = Zeilenzahl
; CH = Eckzeile links oben, CL = Eckspalte
; DH = Eckzeile unten rechts, DL = Eckspalte
; Es erscheint ein inverses Fenster auf dem
; Screen
;
;           MOV  AX,0700      ; down scroll, clear
;                               ; window
;           MOV  BH,F0        ; Attribut invers+
;                               ; blinkend
;           MOV  CX,030F      ; linke obere Ecke
;           MOV  DX,1040      ; rechte untere Ecke
;           INT  10           ; BIOS Routine rufen
;
; Positioniere den Cursor in das Fensters
; AH 02H, BH = Bildschirmseite, DL = Spalte
; DH = Zeile
;
;           MOV  AH,02        ; set cursor
;           MOV  BH,00        ; Seite 0
;           MOV  DX,091B      ; Spalte/Zeile
;           INT  10           ; BIOS Routine rufen
;
; Schreibe String auf dem Schirm
;
```

```

        CALL 200          ; Ausgabe
;
; Rückkehr zu MS-DOS
;
        MOV  AX,4C00      ; DOS-Code "Exit"
        INT 21            ; Terminiere Programm
; hier muß eine Leerzeile kommen
A 200
;=====
; Unterprogramm zur Ausgabe eines Textes
;=====
; Output:
        MOV  DX,208       ; lade Stringadresse
        MOV  AH,09        ; DOS-Code "Write
                          ; String"
        INT 21            ; String ausgeben
        RET              ; Ende Unterprogramm
;=====
; Datenbereich mit dem Textstring
;=====
DB "Der Toolbox Assemblerkurs",0D,0A,"$"
;
; Steueranweisungen für DEBUG
N INT1.COM
R CX
250
W
Q

```

Listing 2.18: Demonstration des INT-Befehls

Mit den JMP-, CALL- und INT-Befehlen steht das Rüstzeug für die Erstellung übersichtlicher Programme zur Verfügung. Beispiele für den Einsatz lernen Sie in den folgenden Abschnitten noch zur Genüge kennen.

## 2.11 Befehle zur Konstruktion von Schleifen

Neben den JMP- und CALL-Befehlen kennt der 8086-Prozessor weitere Anweisungen zur Kontrolle des Programmablaufes. Die Konstruktion von Schleifen bildet dabei ein wichtiges Feld. Aus Hochsprachen sind Konstruktionen wie:

```

REPEAT .... UNTIL ()
DO WHILE ()... END

```

bekannt. Anweisungen zur Erzeugung solcher Schleifen lassen sich auch im 8086-Befehlssatz finden. Die LOOP-Befehle benutzen dabei das Register CX als Zähler und können SHORT-Sprünge über die Distanz von + 127 und -128 Byte ausführen.

### 2.11.1 Der LOOP-Befehl

Der Befehl besitzt die Syntax:

LOOP SHORT Label

Bei jeder Ausführung wird der Inhalt des Registers CX um 1 decrementiert (erniedrigt). Ist der Wert des Register CX ungleich 0, dann verzweigt der Prozessor zum angegebenen SHORT-Label. Andernfalls wird die auf den LOOP-Befehl folgende Adresse ausgeführt. Die folgende kleine Sequenz zeigt schematisch den Einsatz des LOOP-Befehls zur Konstruktion einer REPEAT-UNTIL-Schleife.

```
MOV CX,0005      ; Zähler laden
Start:           ; Schleifenanfang
.
.
.
LOOP Start       ; Schleifenende
.
```

Der Inhalt von CX wird vor der Schleife mit dem Startwert geladen. Anschließend führt der Prozessor die Befehle innerhalb der Schleife n mal aus. Der LOOP-Befehl beeinflusst die Flags nicht.

### 2.11.2 Der LOOPE/LOOPZ-Befehl

Der Befehl besitzt die Syntax:

LOOPE SHORT Label

LOOPZ SHORT Label

und funktioniert ähnlich dem LOOP-Befehl. Der Wert des Registers CX wird zuerst um 1 decrementiert. Die Verzweigung erfolgt, falls die Bedingung:

$CX \neq 0$  und Zero Flag = 1

erfüllt ist. Andernfalls wird die auf den LOOPE/LOOPZ-Befehl folgende Anweisung ausgeführt. Das Zero-Flag kann durch eine vorhergehende Anweisung gesetzt oder gelöscht worden sein. Der Startwert in CX spezifiziert wie oft die Schleife maximal durchlaufen werden darf. Ist das Zero-Flag vorher auf 0 gesetzt, wird die Schleife sofort beendet. Das Label darf nur als SHORT angegeben werden. Die beiden Bezeichnungen LOOPE (Loop While Equal) und LOOPZ (Loop While Zero) erzeugen den gleichen Befehlscode, es handelt sich also nur um einen Befehl mit zwei Namen. Dies ist beim Disassemblieren mit DEBUG zu beachten, da dann nur ein Mnemonic ausgegeben wird. Nachfolgendes kleine Beispiel zeigt die Verwendung des LOOPE/ LOOPZ-Befehls:

```

MOV CX,0005      ; Schleife maximal 5 mal
Start:           ; Schleifenanfang
.
MOV AL,[3000]    ; lade Zeichen
CMP AL, 00       ; Zeichen 00 ?
; terminiere Schleife nach der 5. Abfrage, oder
; falls das Zeichen 00 ist.
    LOOP Start   ; Schleifenende
.

```

Durch den CMP-Befehl wird das Zero-Flag beeinflusst. Ist AL in unserem Beispiel auf 00 gesetzt, terminiert die Schleife unabhängig vom Wert in CX. Der LOOPE/LOOPZ-Befehl verändert selbst keine Flags.

### 2.11.3 Der LOOPNE/LOOPNZ-Befehl

Der Befehl besitzt die Syntax:

```

LOOPNE SHORT Label
LOOPNZ SHORT Label

```

und funktioniert ähnlich dem LOOPE/LOOPZ-Befehl. Der Wert des Registers CX wird zuerst um 1 decrementiert. Die Verzweigung erfolgt, falls die Bedingung:

CX <> 0    und    Zero Flag = 0

erfüllt ist. Andernfalls wird die auf den LOOPNE/ LOOPNZ-Befehl folgende Anweisung ausgeführt. Das Zero-Flag kann durch eine vorhergehende Anweisung gesetzt oder gelöscht worden sein. Der Startwert in CX spezifiziert wird oft die Schleife maximal durchlaufen werden darf. Ist das Zero-Flag vorher auf 1 gesetzt, wird die Schleife sofort beendet. Das Label darf nur als SHORT angegeben werden. Die beiden Bezeichnungen LOOPNE (Loop While Not Equal) und LOOPNZ (Loop While Not Zero) erzeugen den gleichen Befehlscode, es handelt sich also nur um einen Befehl mit zwei Namen. Dies ist beim Disassemblieren mit DEBUG zu beachten, da dann nur ein Mnemonic ausgegeben wird. Nachfolgendes kleine Beispiel zeigt die Verwendung des LOOPNE/ LOOPNZ-Befehls:

```

MOV CX,0005      ; Schleife maximal 5 mal
Start:           ; Schleifenanfang
.
MOV AL,[3000]    ; lade Zeichen
CMP AL, 41       ; Zeichen <> 'A' ?
; terminiere Schleife nach der 5. Abfrage, oder
; falls das Zeichen <> 'A' ist.
    LOOP Start   ; Schleifenende
.

```



Durch den CMP-Befehl wird das Zero-Flag beeinflusst. Ist AL in unserem Beispiel auf 42 gesetzt, terminiert die Schleife unabhängig vom Wert in CX. Der LOOPNE/LOOPNZ-Befehl verändert selbst keine Flags.

## 2.12 Die String-Befehle

Die 80X86-Prozessorfamilie besitzt einen Satz von 5 Befehlen zur Bearbeitung von Strings (Byte- oder Wordfolgen) mit einer Länge von 1 Byte bis 64 KByte. Die Adressierung der Strings erfolgt über die Register DS:SI (Quelle) und ES:DI (Ziel). Die Register SI und DI werden nach Ausführung des Befehls um den Wert 1 erhöht oder erniedrigt um das folgende Stringelement zu adressieren. Die Richtung (increment oder decrement) wird durch den Wert des Direction-Flag (s. Befehlsbeschreibung) bestimmt.

### 2.12.1 Die REPEAT-Anweisungen

Diese Anweisungen werden zusammen mit den String-Befehlen verwendet um die Autoincrement / -decrement-Funktion zu aktivieren. Dadurch lassen sich komplette Strings bearbeiten. Die Mnemonics für die REPEAT-Befehle lauten:

REP	(Repeat)
REPE	(Repeat While Equal)
REPZ	(Repeat While Zero)
REPNE	(Repeat While Not Equal)
REPNZ	(Repeat While Not Zero)

und sind als Prefix direkt vor dem String-Befehl zu platzieren. Die CPU wertet dann den Inhalt des CX-Registers aus und wiederholt den nachfolgenden String-Befehl solange, bis der Wert des Registers 0 annimmt.

### 2.12.2 Die MOVS-Anweisungen (Move-String)

Mit diesen Anweisungen lassen sich Bytes oder Worte innerhalb des Speichers transferieren. Es werden dabei zwei verschiedene Befehle mit der Syntax:

MOVSB	; Move String Byte
MOVSW	; Move String Word

unterschieden. Dabei wird die Adresse des Quellstrings durch die Register DS:SI (Datensegment:Sourceindex) angegeben. Das Byte oder Word wird zum Zielstring kopiert. Dieser wird durch die Register ES:DI (Extrasegment:Destinationindex) adressiert. Nach Ausführung des Befehls zeigen SI und DI auf das folgende String-

element. Durch Kombination mit der REP-Anweisung läßt sich ein ganzer Speicherbereich verschieben.

### 2.12.3 Die CMPS-Anweisung (Compare String)

Mit dieser Anweisung lassen sich zwei Speicherzellen (Byte oder Word) vergleichen. Dabei wird das Zielelement vom Quellelement subtrahiert. Der Befehl verändert die Flags: AF, CF, OF, PF, SF in Abhängigkeit vom Ergebnis. Die Operanden werden allerdings nicht verändert. Nach der Befehlsausführung zeigen DS:SI und ES:DI auf das nächste Stringelement. Durch Kombination mit der REPE/REPZ-Anweisung lassen sich zwei Speicherbereiche vergleichen. Das Register CX ist mit der Stringlänge zu laden. Der REPE/REPZ-Befehl wird solange wiederholt, wie  $CX \neq 0$  (compare while not end of string) ist und die Strings gleich (while strings are equal) sind.

### 2.12.4 Die SCAS-Anweisung (Scan String)

Mit dieser Anweisung wird das durch ES:DI adressierte Byte oder Wort vom Inhalt des Registers AL oder AX subtrahiert. Der Wert des Registers AL oder AX und des Strings bleibt dabei aber unverändert. Lediglich die Flags: AF, CF, OF, SF, PF, SF und ZF werden in Abhängigkeit vom Ergebnis gesetzt. Das Register DI zeigt nach der Ausführung des Befehls auf das folgende Stringelement. Mit dem Befehl läßt sich prüfen, ob ein Wert im String mit dem Inhalt des Registers AL oder AX übereinstimmt. Durch Kombination mit der REPE/REPNE/REPZ/ REPNZ-Anweisung lassen sich komplette Speicherbereiche auf ein Zeichen absuchen. Das Register CX ist mit der Stringlänge zu laden. Der REPNE/REPNZ-Befehl wird solange wiederholt, wie  $CX \neq 0$  (compare while not end of string) ist und der Stringwert gleich dem Wert im Akkumulator (while strings are not equal to scan value) ist. Bei REPE/REPZ wird die Suche solange fortgesetzt, wie die Bedingung (while strings are equal to scan value) erfüllt ist. In beiden Fällen wird das Zero Flag ausgewertet.

### 2.12.5 Die LODS-Anweisung (Load String)

Mit dieser Anweisung wird das durch DS:SI adressierte Byte oder Wort in das Register AL oder AX geladen. Der Befehl verändert keine Flags. Das Register SI zeigt nach der Ausführung des Befehls auf das folgende Stringelement. Der Befehl läßt sich nicht mit den REPEAT-Befehlen nutzen, da jeweils der Wert des Akkumulators überschrieben würde. Ein Einsatz in Softwareschleifen ist aber jederzeit möglich.

### 2.12.6 Die STOS-Anweisung (Store String)

Mit dieser Anweisung wird das durch ES:DI adressierte Byte oder Wort mit dem Inhalt des Registers AL oder AX überschrieben. Der Befehl verändert keine Flags,

setzt aber das Register DI nach der Ausführung auf die Adresse des folgenden String-elements. Der Befehl läßt sich zusammen mit den REPEAT-Anweisungen recht elegant zur Initialisierung von kompletten Datenbereichen benutzen.

### **2.12.7 Der HLT-Befehl**

Dieser Befehl veranlaßt, daß die CPU in den HALT-Modus geht. Damit werden keine neuen Befehle mehr ausgeführt. Der Mode läßt sich durch einen Reset oder einen Hardwareinterrupt beenden.

### **2.12.8 Der LOCK-Befehl**

Dieser Befehl wirkt als Prefix (z.B. bei XCHG) und signalisiert einem Koprozessor, daß der folgende Befehl nicht unterbrochen werden darf.

### **2.12.9 Der WAIT-Befehl**

Dieser Befehl bringt die CPU in den WAIT-Mode, falls die Leitung TEST nicht aktiv ist.

### **2.12.10 Der ESC-Befehl**

Der Befehl leitet einen Opcode ein, der durch einen externen Prozessor (z.B. 8087) bearbeitet wird.

