

# Einführung in die Assemblerprogrammierung mit x86-Prozessoren

**von Günter Born**

**Günter Born**

**Einführung in die  
Assemblerprogrammierung  
für 80x86-Prozessoren**

---

G. Born

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz gemacht. Eventuell vorkommende Warennamen werden benutzt, ohne daß ihre freie Verwendbarkeit gewährleistet werden kann.

Das Buch wurde mit größter Sorgfalt erstellt und korrigiert. Dennoch können Fehler und Ungenauigkeiten nicht ausgeschlossen werden - wir sind auch nur Menschen. Weder Verlag noch Autor können für fehlerhafte Angaben oder gar deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Über Verbesserungsvorschläge, Hinweise auf Fehler und jedwede qualifizierte Kritik freuen wir uns aber.

Alle Rechte, auch der fotomechanischen Wiedergabe und der Veröffentlichung in elektronischen oder sonstigen Medien, behalten wir uns vor. Dieses elektronische Buch darf nur wie ein anderes Buch gehandhabt werden. Die Besitzer der CD-ROM dürfen einen Ausdruck davon erstellen. Die CD-ROM darf nur auf einem PC von einer Person gleichzeitig benutzt werden. Eine Nutzung in einem Netzwerk ist nicht statthaft. Die gewerbliche Nutzung der in diesem Buch gezeigten Beispiele, Modelle, Abbildungen und Ideen ist untersagt.

© 1995 by Günter Born & Addison Wesley (Deutschland) GmbH

Satz: Günter Born

Herstellung: Günter Born & Addison Wesley



---

# Inhaltsverzeichnis

<b>VORWORT</b>	<b>15</b>
<b>EINLEITUNG</b>	<b>17</b>
Für wen ist dieses Buch?	17
Was dieses Buch nicht kann	17
Wie arbeite ich mit diesem Buch?	18
<b>1 GRUNDLAGEN</b>	<b>21</b>
<b>1.1 Wozu braucht man einen Assembler?</b>	<b>21</b>
<b>1.2 Das Hexadezimal-, Binär- und Dezimalsystem</b>	<b>22</b>
<b>1.3 Die logischen Operatoren AND, OR, XOR und NOT</b>	<b>24</b>
<b>2 EINFÜHRUNG IN DEN 8086-BEFEHLSSATZ</b>	<b>27</b>
<b>2.1 Einführung in die 8086-Architektur</b>	<b>27</b>
2.1.1 Die Universalregister	28
2.1.2 Die Index- und Pointer-Register	29
2.1.3 Die Flags	31
2.1.4 Die Segmentregister	32
<b>2.2 Die Befehle der 8086-CPU</b>	<b>38</b>
<b>2.3 Die 8086-Befehle zum Datentransfer</b>	<b>39</b>
2.3.1 Der MOV-Befehl	39
<b>2.4 Der PUSH-Befehl</b>	<b>54</b>
2.4.1 PUSHF, ein spezieller PUSH-Befehl	56
2.4.2 Der POP-Befehl	56
2.4.3 Der POPF-Befehl	58
2.4.4 Der IN-Befehl	62
2.4.5 Der OUT-Befehl	63
2.4.6 Der XCHG-Befehl	64
2.4.7 Der NOP-Befehl	67
2.4.8 Der XLAT-Befehl	67
2.4.9 Der Befehl LEA	69

2.4.10 Die Befehle LDS und LES	70
2.4.11 Die Befehle LAHF und SAHF	72
<b>2.5 Befehle zur Bitmanipulation</b>	<b>73</b>
2.5.1 Der NOT-Befehl	73
2.5.2 Der AND-Befehl	73
2.5.3 Der OR-Befehl	76
2.5.4 Der XOR-Befehl	77
2.5.5 Der TEST-Befehl	79
<b>2.6 Die Shift-Befehle</b>	<b>80</b>
2.6.1 Die Befehle SHL /SAL	81
2.6.2 Der Befehl SHR	82
2.6.3 Der Befehl SAR	83
<b>2.7 Die Rotate-Befehle</b>	<b>84</b>
2.7.1 Der ROL-Befehl	84
2.7.2 Der ROR-Befehl	86
2.7.3 Der RCL-Befehl	86
2.7.4 Der RCR-Befehl	87
<b>2.8 Befehle zur Kontrolle der Flags</b>	<b>87</b>
2.8.1 Clear Carry-Flag (CLC)	87
2.8.2 Complement Carry-Flag (CMC)	88
2.8.3 Set Carry-Flag (STC)	88
2.8.4 Clear Direction-Flag (CLD)	88
2.8.5 Set Direction-Flag (STD)	88
2.8.6 Clear Interrupt-Enable-Flag (CLI)	88
2.8.7 Set Interrupt-Enable-Flag (STI)	89
<b>2.9 Die Arithmetik-Befehle</b>	<b>89</b>
2.9.1 Die Datenformate des 8086-Prozessors	89
2.9.2 Der ADD-Befehl	92
2.9.3 Der ADC-Befehl	93
2.9.4 Der DAA-Befehl	95
2.9.5 Der AAA-Befehl	96
2.9.6 Der SUB-Befehl	96
2.9.7 Der SBB-Befehl	97
2.9.8 Der DAS-Befehl	99
2.9.9 Der AAS-Befehl	99
2.9.10 Der MUL-Befehl	99
2.9.11 Der IMUL-Befehl	101
2.9.12 Der AAM-Befehl	102
2.9.13 Der DIV-Befehl	102
2.9.14 Der IDIV-Befehl	103
2.9.15 Der AAD-Befehl	104
2.9.16 Der CMP-Befehl	105
2.9.17 Der INC-Befehl	106
2.9.18 Der DEC-Befehl	107

---

2.9.19 Der NEG-Befehl	108
2.9.20 Der CBW-Befehl	109
2.9.21 Der CWD-Befehl	109
<b>2.10 Die Programmtransfer-Befehle</b>	<b>110</b>
2.10.1 Die JMP-Befehle	110
2.10.2 Die CALL-Befehle	135
2.10.3 Der INT-Befehl	141
<b>2.11 Befehle zur Konstruktion von Schleifen</b>	<b>144</b>
2.11.1 Der LOOP-Befehl	145
2.11.2 Der LOOPE/LOOPZ-Befehl	145
2.11.3 Der LOOPNE/LOOPNZ-Befehl	146
<b>2.12 Die String-Befehle</b>	<b>147</b>
2.12.1 Die REPEAT-Anweisungen	147
2.12.2 Die MOVS-Anweisungen (Move-String)	147
2.12.3 Die CMPS-Anweisung (Compare String)	148
2.12.4 Die SCAS-Anweisung (Scan String)	148
2.12.5 Die LODS-Anweisung (Load String)	148
2.12.6 Die STOS-Anweisung (Store String)	148
2.12.7 Der HLT-Befehl	149
2.12.8 Der LOCK-Befehl	149
2.12.9 Der WAIT-Befehl	149
2.12.10 Der ESC-Befehl	149
<b>3 EINFÜHRUNG IN DEN A86</b>	<b>151</b>
<b>3.1 Einführung in die 8086-Architektur</b>	<b>152</b>
3.1.1 Die Universalregister	153
3.1.2 Die Index- und Pointer-Register	154
3.1.3 Die Flags	155
3.1.4 Die Segmentregister	157
<b>3.2 Die 8086-Befehle zum Datentransfer</b>	<b>163</b>
3.2.1 Der MOV-Befehl	163
3.2.2 Der PUSH-Befehl	176
3.2.3 Der POP-Befehl	178
3.2.4 Der IN-Befehl	183
3.2.5 Der OUT-Befehl	184
3.2.6 Der XCHG-Befehl	185
3.2.7 Der NOP-Befehl	187
3.2.8 Der XLAT-Befehl	188
3.2.9 Der Befehl LEA	188
3.2.10 Die Befehle LDS und LES	189
3.2.11 Die Befehle LAHF und SAHF	191

<b>3.3 Befehle zur Bitmanipulation</b>	<b>192</b>
3.3.1 Der NOT-Befehl	192
3.3.2 Der AND-Befehl	193
3.3.3 Der OR-Befehl	195
3.3.4 Der XOR-Befehl	197
3.3.5 Der TEST-Befehl	198
<b>3.4 Die Shift-Befehle</b>	<b>199</b>
3.4.1 Die Befehle SHL /SAL	200
3.4.2 Der Befehl SHR	201
3.4.3 Der Befehl SAR	202
<b>3.5 Die Rotate-Befehle</b>	<b>203</b>
3.5.1 Der ROL-Befehl	203
3.5.2 Der ROR-Befehl	204
3.5.3 Der RCL-Befehl	205
3.5.4 Der RCR-Befehl	206
<b>3.6 Befehle zur Kontrolle der Flags</b>	<b>206</b>
3.6.1 Clear Carry-Flag (CLC)	206
3.6.2 Complement Carry-Flag (CMC)	207
3.6.3 Set Carry-Flag (STC)	207
3.6.4 Clear Direction-Flag (CLD)	207
3.6.5 Set Direction-Flag (STD)	207
3.6.6 Clear Interrupt-Enable-Flag (CLI)	207
3.6.7 Set Interrupt-Enable-Flag (STI)	208
<b>3.7 Die Arithmetik-Befehle</b>	<b>208</b>
3.7.1 Die Datenformate des 8086-Prozessors	208
3.7.2 Der ADD-Befehl	211
3.7.3 Der ADC-Befehl	212
3.7.4 Der DAA-Befehl	213
3.7.5 Der AAA-Befehl	215
3.7.6 Der SUB-Befehl	215
3.7.7 Der SBB-Befehl	216
3.7.8 Der DAS-Befehl	217
3.7.9 Der AAS-Befehl	218
3.7.10 Der MUL-Befehl	218
3.7.11 Der IMUL-Befehl	219
3.7.12 Der AAM-Befehl	220
3.7.13 Der DIV-Befehl	221
3.7.14 Der IDIV-Befehl	222
3.7.15 Der AAD-Befehl	223
3.7.16 Der CMP-Befehl	223
3.7.17 Der INC-Befehl	224
3.7.18 Der DEC-Befehl	225
3.7.19 Der NEG-Befehl	227
3.7.20 Der CBW-Befehl	227
3.7.21 Der CWD-Befehl	228

---

<b>3.8 Die Programmtransfer-Befehle</b>	<b>228</b>
3.8.1 Die JMP-Befehle	228
3.8.2 Die CALL-Befehle	242
3.8.3 Der INT-Befehl	244
<b>3.9 Befehle zur Konstruktion von Schleifen</b>	<b>246</b>
3.9.1 Der LOOP-Befehl	246
3.9.2 Der LOOPE/LOOPZ-Befehl	247
3.9.3 Der LOOPNE/LOOPNZ-Befehl	247
<b>3.10 Die String-Befehle</b>	<b>248</b>
3.10.1 Die REPEAT-Anweisungen	248
3.10.2 Die MOVS-Anweisungen (Move-String)	248
3.10.3 Die CMPS-Anweisung (Compare String)	249
3.10.4 Die SCAS-Anweisung (Scan String)	249
3.10.5 Die LODS-Anweisung (Load String)	249
3.10.6 Die STOS-Anweisung (Store String)	249
<b>3.11 Der HLT-Befehl</b>	<b>250</b>
<b>3.12 Der LOCK-Befehl</b>	<b>250</b>
<b>3.13 Der WAIT-Befehl</b>	<b>250</b>
<b>3.14 Der ESC-Befehl</b>	<b>250</b>
<b>3.15 Die NEC V20/V30-Befehle</b>	<b>250</b>
<b>3.16 Die 80186/80286 Befehlserweiterungen</b>	<b>253</b>
<b>3.17 Die 80x87-Fließkommabefehle</b>	<b>257</b>
<b>4 DIE A86-DIRECTIVEN</b>	<b>259</b>
<b>4.1 Die Darstellung von Konstanten</b>	<b>259</b>
<b>4.2 Die RADIX-Directive</b>	<b>259</b>
<b>4.3 Der HIGH/LOW-Operator</b>	<b>260</b>
<b>4.4 Der BY-Operator</b>	<b>260</b>
<b>4.5 Operationen auf Ausdrücken</b>	<b>260</b>
4.5.1 Addition	261
4.5.2 Subtraktion	261
4.5.3 Multiplikation und Division	261
4.5.4 Schiebeoperatoren	261
4.5.5 Die logischen Operatoren	262
4.5.6 Der NEG-Operator	262

4.5.7 Die Vergleichsoperatoren	262
4.5.8 Stringvergleiche	263
4.5.9 Definition von Datenbereichen	263
4.5.10 SHORT/LONG/NEAR	263
4.5.11 Der OFFSET-Operator	263
4.5.12 Der TYPE-Operator	264
4.5.13 Der THIS-Operator	264
<b>4.6 Anweisungen zur Segmentierung</b>	<b>265</b>
4.6.1 CODE SEGMENT	265
4.6.2 DATA SEGMENT	266
4.6.3 Die ORG-Directive	266
4.6.4 Die EVEN-Directive	267
<b>4.7 Datendefinitionen mit DB, DW, DD, DQ, DT</b>	<b>267</b>
<b>4.8 Die Definition von Strukturen</b>	<b>269</b>
<b>4.9 Vorwärtsreferenzen</b>	<b>269</b>
<b>4.10 Die EQU-Directive</b>	<b>270</b>
<b>4.11 Die PROC-Directive</b>	<b>272</b>
<b>4.12 Die LABEL-Directive</b>	<b>273</b>
<b>4.13 Die NAME-Directive</b>	<b>273</b>
<b>4.14 Die PUBLIC-Directive</b>	<b>274</b>
<b>4.15 Die EXTRN-Directive</b>	<b>274</b>
<b>4.16 Die MAIN-Directive</b>	<b>275</b>
<b>4.17 Die END-Directive</b>	<b>276</b>
<b>4.18 Die SEGMENT-Directive</b>	<b>276</b>
<b>4.19 CODE-, DATA- und STACK-Directiven</b>	<b>278</b>
<b>4.20 Die ENDS-Directive</b>	<b>278</b>
<b>4.21 Die GROUP-Directive</b>	<b>278</b>
<b>4.22 Die SEG-Directive</b>	<b>279</b>
<b>4.23 Makros und bedingte Assemblierung</b>	<b>279</b>
4.23.1 Die impliziten Makros des A86	279
4.23.2 Die expliziten Makros	283

---

4.23.3 Schleifen in Makros	286
<b>4.24 Bedingte Assemblierung</b>	<b>290</b>
<b>4.25 Assemblierung und Linken</b>	<b>291</b>
4.25.1 Erzeugung von COM-Dateien	292
4.25.2 Erzeugung von BIN-Dateien	293
4.25.3 Erzeugung von OBJ-Dateien	293
4.25.4 Die A86-Optionen	293
<b>4.26 Linken von OBJ-Dateien</b>	<b>295</b>
<b>5 PROGRAMMENTWICKLUNG MIT MASM 6.X</b>	<b>303</b>
<b>5.1 Die Darstellung von Konstanten</b>	<b>305</b>
<b>5.2 Die RADIX-Directive</b>	<b>305</b>
<b>5.3 Definition von Datenbereichen</b>	<b>306</b>
5.3.1 Datendefinitionen mit DB, DW, DD, DQ und DT	306
5.3.2 Der DUP-Operator	307
5.3.3 Der LENGTHOF-Operator	309
5.3.4 Der SIZEOF-Operator	309
5.3.5 Der TYPE-Operator	310
5.3.6 Die Definition von Strukturen	310
<b>5.4 Die MODEL-Directive</b>	<b>310</b>
<b>5.5 Die Segmentanweisung</b>	<b>311</b>
<b>5.6 Die ORG-Anweisung</b>	<b>312</b>
<b>5.7 Der OFFSET Operator</b>	<b>312</b>
<b>5.8 Die Segment-Override-Anweisung</b>	<b>314</b>
<b>5.9 Die EQU Directive</b>	<b>315</b>
<b>5.10 Operationen auf Ausdrücken</b>	<b>316</b>
5.10.1 Addition	317
5.10.2 Subtraktion	317
5.10.3 Multiplikation und Division	317
5.10.4 Schiebeoperatoren (SHL, SHR)	318
5.10.5 Logische Operatoren	318
5.10.6 Vergleichsoperatoren	318
<b>5.11 Die EVEN Directive</b>	<b>319</b>

<b>5.12 Die PROC Directive</b>	<b>323</b>
5.12.1 Erstellen einer EXE-Datei aus mehreren OBJ-Files	324
5.12.2 Die Directiven für externe Module	331
<b>5.13 Einbinden von Assemblerprogrammen in Hochsprachen</b>	<b>333</b>
<b>6 PROGRAMMENTWICKLUNG MIT TASM</b>	<b>341</b>
<b>6.1 Die Darstellung von Konstanten</b>	<b>343</b>
<b>6.2 Die RADIX-Directive</b>	<b>343</b>
<b>6.3 Definition von Datenbereichen</b>	<b>344</b>
6.3.1 Datendefinitionen mit DB, DW, DD, DQ und DT	344
<b>6.4 Der DUP-Operator</b>	<b>345</b>
6.4.2 Die Definition von Strukturen	347
6.4.3 Die MODEL-Directive	347
<b>6.5 Die Segmentanweisung</b>	<b>348</b>
6.5.1 Die ORG-Anweisung	349
<b>6.6 Der OFFSET Operator</b>	<b>349</b>
<b>6.7 Die Segment-Override-Anweisung</b>	<b>351</b>
<b>6.8 Die EQU Directive</b>	<b>352</b>
<b>6.9 4.10 Operationen auf Ausdrücken</b>	<b>354</b>
6.9.1 Addition	354
6.9.2 Subtraktion	354
6.9.3 Multiplikation und Division	354
6.9.4 Schiebeoperatoren (SHL, SHR)	355
6.9.5 Logische Operatoren	355
<b>6.10 Vergleichsoperatoren</b>	<b>356</b>
<b>6.11 Die EVEN Directive</b>	<b>356</b>
<b>6.12 Die PROC-Directive</b>	<b>360</b>
6.12.1 Anmerkungen	361
<b>6.13 Erstellen einer EXE-Datei aus mehreren OBJ-Files</b>	<b>362</b>
<b>6.14 Die Directiven für externe Module</b>	<b>369</b>
6.14.1 Die PUBLIC-Directive	369
6.14.2 Die EXTRN-Directive	369
6.14.3 Die END-Directive	370
6.14.4 Die GROUP-Directive	370

---

<b>6.15 Einbinden von Assemblerprogrammen in Hochsprachen</b>	<b>371</b>
<b>7 DEBUG ALS ASSEMBLER UND WERKZEUG IN DOS.</b>	<b>377</b>
<b>7.1 Die Funktion des Debuggers</b>	<b>377</b>
7.1.1 Der Programmstart	377
7.1.2 Der DUMP-Befehl	379
7.1.3 Die ENTER-Funktion	380
7.1.4 Der FILL-Befehl	382
7.1.5 Die MOVE-Funktion	383
7.1.6 Die INPUT-/OUTPUT-Befehle	383
7.1.7 Die HEX-Funktion	384
7.1.8 Die COMPARE-Funktion	385
7.1.9 Die SEARCH-Funktion	385
7.1.10 Der REGISTER-Befehl	386
7.1.11 Die File I/O-Befehle	387
7.1.12 Der NAME-Befehl	388
7.1.13 Der WRITE-Befehl	388
7.1.14 Der LOAD-Befehl	389
7.1.15 Programmentwicklung mit DEBUG	389
7.1.16 Der ASSEMBLE-Befehl	390
7.1.17 Assemblierung aus einer Datei	392
7.1.18 Der UNASSEMBLE-Befehl	395
7.1.19 Programmtests mit DEBUG	397
7.1.20 Der GO-Befehl	397
7.1.21 Der TRACE-Befehl	398
7.1.22 Der PROCEED-Befehl	398
7.1.23 Die Expanded-Memory-Befehle	399
7.1.24 Anmerkungen zu DR-DOS 5.0/6.0	399
<b>ANHANG A</b>	<b>401</b>
<b>ANHANG B</b>	<b>403</b>
<b>ANHANG C</b>	<b>405</b>
<b>Debuggen mit dem D86</b>	<b>405</b>
Der A86-Assembler-Modus [F7]	406
<b>ANHANG D</b>	<b>409</b>
<b>Die Fehlermeldungen des A86</b>	<b>409</b>
<b>ANHANG E</b>	<b>416</b>
<b>Attribute beim Monochromadapter</b>	<b>416</b>

<b>ANHANG F</b>	<b>417</b>
<b>Kodierung Farbattribute bei Colorkarten</b>	<b>417</b>
<b>ANHANG G</b>	<b>418</b>
<b>Die 8086-Assemblerbefehle</b>	<b>418</b>
<b>LITERATUR</b>	<b>421</b>
<b>STICHWORTVERZEICHNIS</b>	<b>423</b>

# Vorwort

Mit dem Microsoft Macroassembler (MASM) und dem Borland Turbo Assembler (TASM) stehen recht leistungsfähige Werkzeuge für die Assemblerprogrammierung zur Verfügung. Eine weitere sehr interessante Alternative bietet der A86-Assembler von Eric Isaacson, der als Shareware vertrieben wird. Der A86 ist zwar nicht ganz zum MASM kompatibel, bietet dafür aber eine Reihe zusätzlicher Optionen. Als Manko für einen Einstieg in die Assemblerprogrammierung erweist sich jedoch häufig die Komplexität dieser Assembler.

Andererseits haben viele Einsteiger in den vergangenen Jahren in Kursen über technische Informatik und in meiner Serie über Assemblerprogrammierung (Zeitschrift Toolbox - DMV Verlag) bestätigt, daß ein Einstieg in die Thematik gar nicht so schwer ist. Bei diesen Einführungskursen habe ich nicht einmal einen Assembler verwendet, daß DOS-Programm DEBUG reichte vollständig aus. Daher entstand die Idee, aus dem Material meiner Assemblerbücher und den bisher gewonnenen Erfahrungen einen Leitfaden für den Einstieg in die Assemblerprogrammierung zu schaffen. Dabei sollte jeder DOS-Anwender ohne zusätzlichen Softwarekauf die Möglichkeit zum Einstieg in die Assemblerprogrammierung haben. In diesem Projekt wurden gänzlich neue Wege beschritten. Dieses Buch liegt in elektronischer Form auf einer CD-ROM vor. Zur Einarbeitung in die grundlegenden Assemblerbefehle läßt sich der DOS-Debugger verwenden. Wer trotzdem direkt mit einem Assembler einsteigen möchte, kann auf den Shareware-Assembler A86 zurückgreifen, der kostenlos auf der CD-ROM beiliegt. Letztendlich wurde aber auch an die Besitzer des MASM oder TASM gedacht. Sie erhalten in diesem Buch eine Einführung in die Assemblerprogrammierung mit diesen Werkzeugen.

In dieser Hinsicht wünsche ich allen Einsteigern viel Spaß und Erfolg beim Einstieg in die Assemblerprogrammierung.

Günter Born



---

# Einleitung

Trotz gestiegener Leistungsfähigkeit heutiger Compiler und der Verfügbarkeit der Sprache C werden immer noch viele Programme in Assembler erstellt. Die Gründe reichen von der fehlenden Möglichkeit aus Hochsprachen auf DOS und die Hardware zuzugreifen, bis hin zur Optimierung spezieller Problemstellungen (z.B. CRC-Generierung). Ein Assemblerprogramm ist in den meisten Fällen wesentlich kürzer als das entsprechende Produkt eines Compilers. Nach dem Motto "totgesagte leben länger" erfreut sich die Programmierung in Assembler einer steigenden Beliebtheit. Auf dem Markt werden neben den Produkten MASM (Microsoft) und TASM (Borland), sowie dem Shareware A86-Assembler auch viele Hochsprachencompiler mit integriertem Assembler angeboten. Beispielhaft seien hier Turbo Pascal und Turbo C von Borland genannt. Für kleinere Problemstellungen reicht sogar der im Lieferumfang von DOS enthaltene Debugger (DEBUG.COM).

Ein Manko für den Einsteiger ist der Umgang mit dem Befehlssatz der 8086-Prozessoren. Hier bieten die Assemblerhandbücher wenig Hilfestellung. Zusatzliteratur ist zwar meist vorhanden, deckt aber nicht alle Bereiche ab. Diesen Mangel versucht das vorliegende Werk zu beheben - enthält es doch neben der Beschreibung der Assembler einen kompletten Einführungsteil in den 80x86-Befehlssatz mit vielen Beispielpogrammen. Zur Bearbeitung dieses Einführungsteils reicht der DOS-Debugger aus. Erfahrenere Nutzer können die Programme später mit MASM oder TASM bearbeiten und damit als OBJ-Files in eigenen Applikationen einbinden. Damit erhält auch der Einsteiger die Möglichkeit, in die Welt der Assemblerprogrammierung einzusteigen.

## Für wen ist dieses Buch?

Dieses Buch wendet sich an alle Personen, die sich in die Assemblerprogrammierung der 8086-Prozessoren einarbeiten möchten. Vermittelt werden die Kenntnisse, um den Befehlssatz der 8086-CPU auf allen 80x86-Rechnern einzusetzen. Als Programmierumgebung wird dabei DOS verwendet. Zur Durchführung der Übungen reicht daher auch der DOS-Debugger. Auf Wunsch läßt sich aber der A86-Assembler zur Übersetzung der Assemblerprogramme benutzen.

## Was dieses Buch nicht kann

Wenn Sie sich bereits in der Assemblerprogrammierung bestens auskennen und nun das Werk mit absoluten Insiderinformationen suchen, wird das vorliegende Buch sicher keine Hilfe sein. Das gleiche gilt, falls Sie unter Windows 3.x oder Windows 95 in Assembler programmieren möchten. Hier handelt es sich um spezielle Themenbereiche, die in den Handbüchern der Softwarehersteller besprochen werden.

Ähnliches gilt, falls Sie Informationen über die Programmierung der 80386/80486-Prozessoren suchen. Die Firma INTEL gibt für diese Prozessoren Handbücher mit den entsprechenden Informationen heraus. Weiterhin enthält die Dokumentation zu MASM und TASM eine Beschreibung dieser Befehle. Für die Programmierung unter DOS und für den Einstieg in die Assemblerprogrammierung sind nur die Basisbefehle des 8086-Prozessors erforderlich.

Dieses Online-Buch hat auch nicht das Ziel, als Ersatzhandbuch für MASM und TASM zu dienen. Sofern Sie diese Produkte offiziell erworben haben, verfügen Sie auch über die zugehörigen Handbücher. Allerdings fehlt dem Einsteiger in diesen Dokumenten häufig der rote Faden. Daher eignen sich die Teile über MASM und TASM für die ersten Schritte in der Assemblerprogrammierung.

## Wie arbeite ich mit diesem Buch?

**Kapitel 1** beschäftigt sich mit den Grundzügen der Assemblerprogrammierung. Wozu braucht man einen Assembler? Was ist eine Hexadezimalzahl, wie wird eine Dezimalzahl in eine Hexadezimalzahl umgerechnet und wie wirken die bool'schen Funktionen. Dies sind Fragestellungen die hier behandelt werden. Sofern Ihnen diese Punkte nicht klar sind, sollten Sie dieses Kapitel lesen.

Der erste Schritt ist immer der schwierigste. Gerade bei Assemblerprogrammen verunsichern die vielen Steueranweisungen den Einsteiger. Er möchte eigentlich die Wirkungsweise der einzelnen Befehle kennen lernen. Wenn Sie einen Einstieg ohne Ballast wünschen, eignet sich **Kapitel 2** sehr gut. Dieses Kapitel bringt eine detaillierte Einführung in den Befehlssatz der 8086-Prozessoren. Dies ist der Mode unter dem DOS auch auf 80286-, 80386- und 80486-Prozessoren betrieben wird. Zum Studium des Kapitels ist noch nicht einmal ein Assembler erforderlich. Alle Beispiele lassen sich mit dem DOS-Programm DEBUG ausführen. Dadurch entfallen alle Steueranweisungen für den Assembler. Weiterhin läßt sich ohne große Kosten prüfen, ob ein Einstieg in die Assemblerprogrammierung lohnt. Nebenbei lernen Sie mit DEBUG kleine Werkzeuge zum täglichen Umgang mit DOS erstellen lassen. Ein Programm zur Abschaltung der NumLock-Taste, Abfrage von Benutzereingaben in Batchdateien, Umschaltung der parallelen Schnittstellen, dies sind nur einige der angesprochenen Themen.

Mit dem A86 steht ein sehr preiswerter Sharewareassembler zur Verfügung. Dieses Programm ist auf der CD-ROM gespeichert. Daher besteht auch die Möglichkeit, den Einstieg in die Assemblerprogrammierung über den A86 zu wagen. **Kapitel 3** enthält nochmals eine Einführung in den 8086-Befehlssatz (analog Kapitel 2). Aber in diesem Kapitel werden alle Befehle aus Sicht des A86-Assemblers behandelt. Diese Einführung läßt sich auch ohne große Änderungen auf den MASM und TASM anwenden.

**Kapitel 4** geht auf die Steueranweisungen und spezielle Fragen im Zusammenhang mit dem A86-Assembler ein. Hier lernen Sie, was die Anweisungen zu Beginn und am Ende des Assemblerprogramms bedeuten. Sie erfahren auch, wie sich OBJ- und EXE-

Dateien erstellen lassen. Die OBJ-Dateien lassen sich dann in Hochspracheprogramme wie PASCAL oder C einbinden.

**Kapitel 5** ist für die Besitzer des Microsoft MASM gedacht. Es bringt dann die Einführung in den Umgang mit diesem Werkzeug. Hier werden die wichtigsten Steueranweisungen behandelt. Die in Kapitel 2 diskutierten Beispiele werden in die MASM-Syntax umgesetzt. Als Ergebnis liegen anschließend lauffähige COM- und EXE-Dateien vor. Den Abschluß bildet die Diskussion der Einbindung von Assemblerprogrammen in Hochsprachen.

Wer den Turbo Assembler besitzt, kann direkt von Kapitel 2 nach **Kapitel 6** springen. Dieses Kapitel bringt dann die Einführung in den Umgang mit diesem Werkzeug. Es werden die wichtigsten Steueranweisungen behandelt und die in Kapitel 2 diskutierten Beispiel in die TASM-Syntax umgesetzt. Als Ergebnis liegen anschließend lauffähige COM- und EXE-Dateien vor. Den Abschluß bildet die Diskussion der Einbindung von Assemblerprogrammen in Hochsprachen.

**Kapitel 7** enthält nochmals eine kurze Einführung in den Umgang mit dem DOS-DEBUG-Programm. Dieses Programm erlaubt, bei richtiger Anwendung, die Erstellung und Übersetzung kleinerer Assemblerprogramme.

Der Anhang enthält weitere Tabellen und einige Beispielprogramme zur Anwendung der erlernten Kenntnisse.



# 1 Grundlagen

## 1.1 Wozu braucht man einen Assembler?

Anders als in BASIC oder anderen interpretativen Sprachen besteht ein ausführbares EXE- oder COM-Programm aus einer Aneinanderreihung von Binärzahlen. Diese Zahlen lassen sich zum Beispiel mit DEBUG als Hexdump auf dem Bildschirm ausgeben:

```
D9 C3 00 43 47 33 45 89 90 90 90 ....
```

Die CPU des Rechners kann nun diese Codes als Steueranweisungen (Maschinenprogramm) verarbeiten. Für den Menschen ist die Darstellung jedoch alles andere als instruktiv. Er benötigt eine symbolische Schreibweise für die Befehle. Aufgabe des Assemblers ist es nun, ein Programm mit symbolischen Assembleranweisungen in einzelne Maschinencodes umzuformen und damit ein Maschinenprogramm in Form einer EXE- oder COM-Datei zu erzeugen. Die Assembleranweisungen werden per Editor in eine Textdatei mit der Extension .ASM abgelegt.

```
— .MODEL TINY
   .STACK 200H
   .DATA           ; Daten vereinbaren
Text DB 'Hallo $'
   .CODE           ; Programm Start
Start:
  MOV AH,09        ; Textausgabe
  MOV DX,Offset Text
  INT 21
  MOV AX,4C00      ; Exit
  INT 21
  END Start
```

Bild 1.1: Aufbau eines Assemblerprogramms

Der Assembler liest dann die Anweisungen der Quelldatei (.ASM) und erzeugt daraus eine COM- oder eine OBJ-Datei, je nach Anweisung des Nutzers. Bei der Übersetzung

führt der Assembler weiterhin noch eine Fehlerüberprüfung und eine Berechnung der Sprungadressen bei JMP- und CALL-Befehlen durch.

Besteht das Programm nur aus Assembleranweisungen und wird die Größe auf 64 KByte begrenzt, besteht bei vielen Assemblern die Möglichkeit direkt eine ausführbare COM-Datei zu erzeugen. Soll das Programm aber mit anderen Programmen oder -teilen zusammen gebunden werden, erzeugt der Assembler eine OBJ-Datei, die nur den übersetzten Code enthält. Diese OBJ-Dateien lassen sich dann durch ein spezielles Programm, welches als Linker bezeichnet wird, zu einer ausführbaren COM- oder EXE-Datei zusammenbinden. Microsoft liefert hierzu das Programm LINK.EXE beim Assembler mit. Bei Borland gibt es ein ähnliches Werkzeug (TLINK.EXE) für diese Aufgabe.

Um ein ausführbares Programm zu erhalten sind in der Regel folgende Schritte erforderlich:

- ◆ Erstellung des Quellprogramms mit einem Texteditor
- ◆ Übersetzung des Quellprogramms mit einem Assembler in eine OBJ- oder COM-Datei
- ◆ Falls OBJ-Dateien vorliegen sind diese mit einem Linker zu einer ausführbaren COM- oder EXE-Datei zu kombinieren

Weitere Einzelheiten lernen Sie in den folgenden Kapiteln kennen.

## 1.2 Das Hexadezimal-, Binär- und Dezimalsystem

Bei der Assemblerprogrammierung werden Zahlen und Konstante häufig im Hexadezimalsystem oder als Binärwerte verarbeitet. Die Umrechnung in Dezimalzahlen ist recht einfach. Die dezimale Zahl 123 setzt sich folgendermaßen zusammen:

$$1 * 100 + 2 * 10 + 3 * 1$$

Bei einer Binärzahl reduziert sich der Wertebereich für eine Ziffer auf 0 und 1. Der Wert 1011<sub>B</sub> läßt sich damit zu:

$$\begin{array}{rcccc}
 1 * 2^{**3} & + & 0 * 2^{**2} & + & 1 * 2^{**1} & + & 1 * 2^{**0} \\
 8 & + & 0 & + & 2 & + & 1 \\
 & & & & & & \mathbf{11 \text{ (dezimal)}}
 \end{array}$$

darstellen. Jede Binärziffer ist mit der entsprechenden Wertigkeit ( $2^{**x}$ ) zu multiplizieren. Das Ergebnis der Addition ergibt den dezimalen Wert. Zur

Rückrechnung der Dezimalzahl in den Binärwert ist eine sukzessive Division durch 2 vorzunehmen:

$$\begin{aligned}
 11 & : 2 = 5 \text{ Rest } 1 \quad (2 * 2^{**0}) \\
 5 & : 2 = 2 \text{ Rest } 1 \quad (2 * 2^{**1}) \\
 2 & : 2 = 1 \text{ Rest } 0 \quad (2 * 2^{**2}) \\
 1 & : 2 = 0 \text{ Rest } 1 \quad (2 * 2^{**3})
 \end{aligned}$$

$$11 = 1011B$$

Der Divisionsrest bestimmt die Ziffern der Binärzahl. Beim Hexadezimalsystem umfaßt der Bereich für eine Ziffer insgesamt 16 Werte:

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
13	D
14	E
15	F

Tabella 1.1: Umrechnung Hex-Dez

Der dezimale Wert 13 läßt sich somit als 0DH darstellen. Eine Umrechnung zwischen binären und hexadezimalen Werten ist recht einfach. Jeder Hexziffer entsprechen 4 binäre Stellen. Der Wert:

$$\begin{array}{cccc}
 3 & 7 & F & D \\
 0011 & 0111 & 1111 & 1011
 \end{array}
 \begin{array}{l}
 \text{H(exadezimal)} \\
 \text{B(inär)}
 \end{array}$$

ist relativ einfach zwischen den Zahlensystemen umrechenbar. Eine Konvertierung zwischen Dezimal- und Hexadezimalsystem erfolgt durch Multiplikation oder Division der Ziffern mit dem Wert 16.

$$\begin{aligned}
 31FH & = 3 * 16^{**2} + 1 * 16^{**1} + 15 * 16^{**0} \\
 & = \quad 768 \quad + \quad 16 \quad + \quad 15 \\
 & = 799 \text{ (dezimal)}
 \end{aligned}$$

Die Rückwandlung einer Dezimalzahl in eine Hexzahl erfolgt durch Division mit 16:

```

69 : 16 = 4 Rest 5  (5 * 16 ** 0)
 4 : 16 = 0 Rest 4  (4 * 16 ** 1)
69      = 45H

```

Bezüglich der Darstellung der Zahlen durch den Prozessor (Byte, Wort, Integer, BCD, etc.) möchte ich auf die nachfolgenden Kapitel verweisen.

## 1.3 Die logischen Operatoren AND, OR, XOR und NOT

Mit diesen Operatoren lassen sich logische Verknüpfungen zwischen zwei Operanden durchführen. Sowohl der 80x86-Befehlssatz als auch die meisten Assembler kennen die logischen Befehle. Bei der AND-Verknüpfung wird jedes Bit der Operanden überprüft. Nur wenn beide Bits gesetzt sind, erhält das Ergebnis den Wert 1.

X1	X2	AND
0	0	0
0	1	0
1	0	0
1	1	1

*Tabelle 1.2: Die AND-Verknüpfung*

Bei der OR-Verknüpfung wird das Ergebnis auf 1 gesetzt, falls eines der Eingangsbits den Wert 1 besitzt.

X1	X2	OR
0	0	0
0	1	1
1	0	1
1	1	1

*Tabelle 1.3: Die OR-Verknüpfung*

Bei der XOR-Verknüpfung wird ebenfalls jedes Bit der Operanden überprüft. Nur wenn beide Bits unterschiedliche Werte besitzen, erhält das Ergebnis den Wert 1.

X1	X2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

*Tabelle 1.4: Die XOR-Verknüpfung*

Bei der NOT-Operation wird einfach der Wert des Eingangsbits im Ergebnis invertiert weitergegeben.

Weitere Hinweise zu diesen Operationen finden sich in den nachfolgenden Kapiteln.



---

## 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.

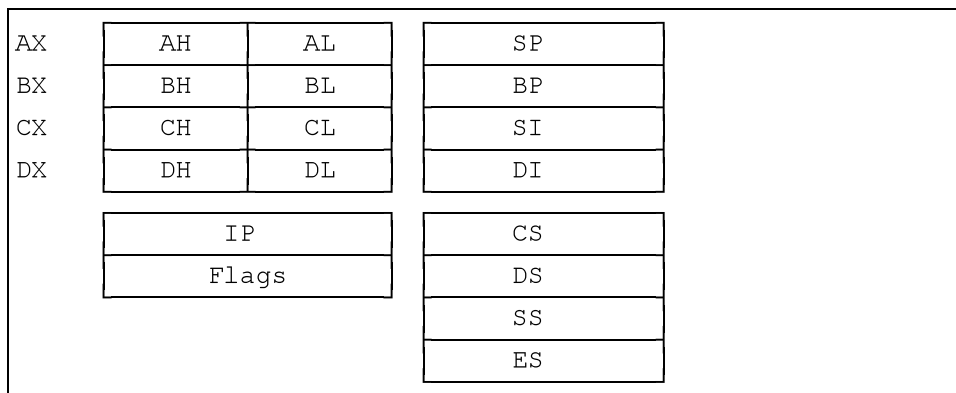


Bild 2.1: Die 8086-Registerstruktur

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.

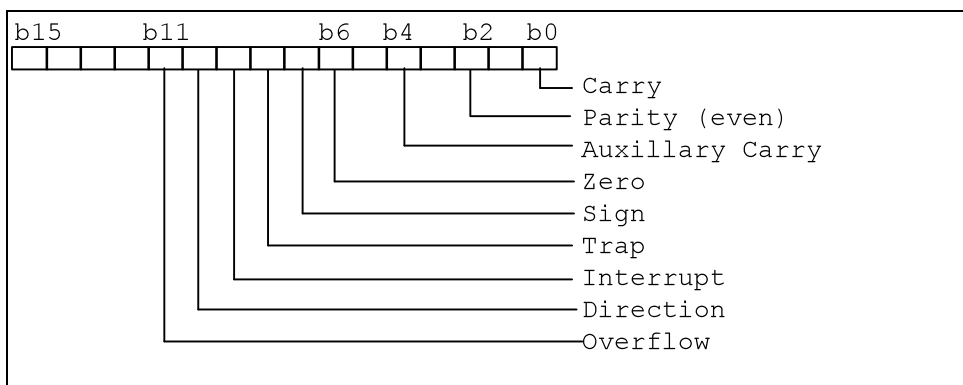


Bild 2.2: Die Kodierung der 8086-Flags

### 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.

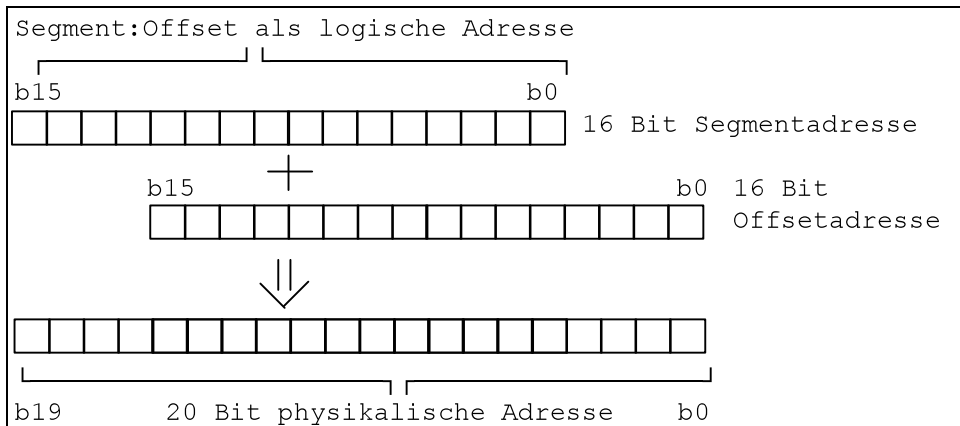


Bild 2.3: Adreßdarstellung in der Segment:Offset-Notation

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.

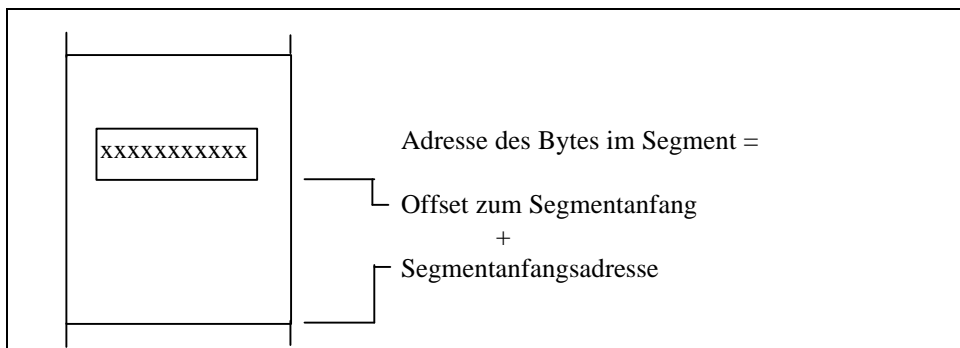


Bild 2.4: Angabe einer Speicheradresse mit Segment:Offset

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
0007  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.

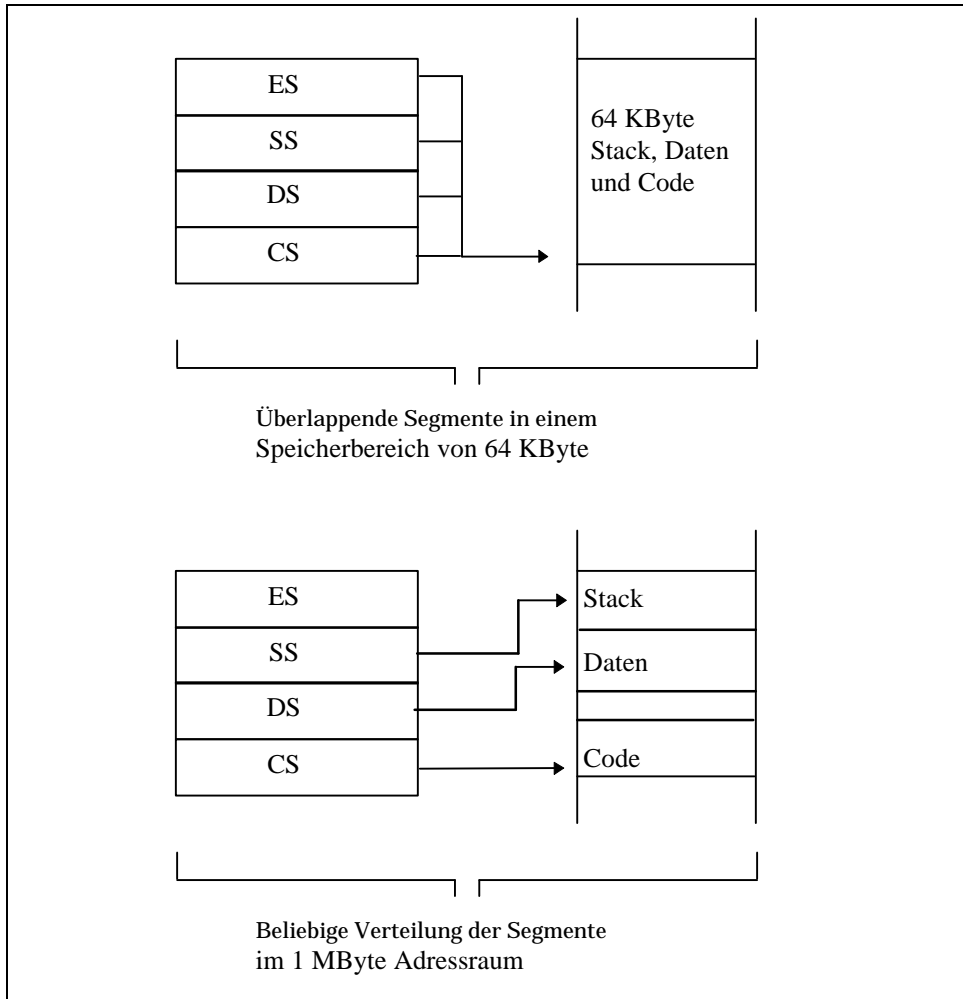


Bild 2.5: Verschiebung der Segmente im Adreßraum

### 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.

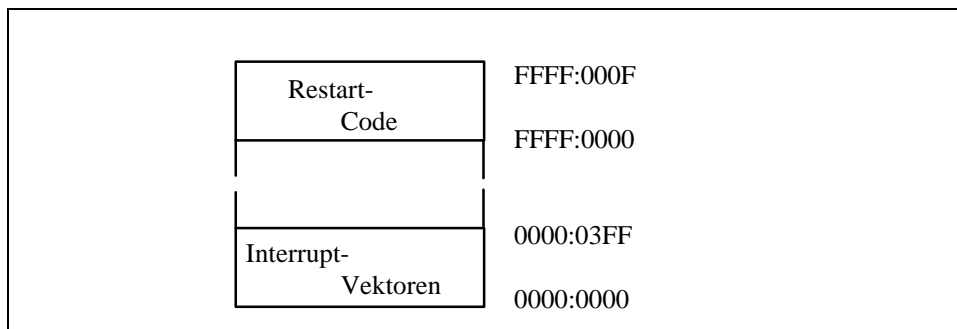


Bild 2.6: Reservierte Adreßbereiche

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).

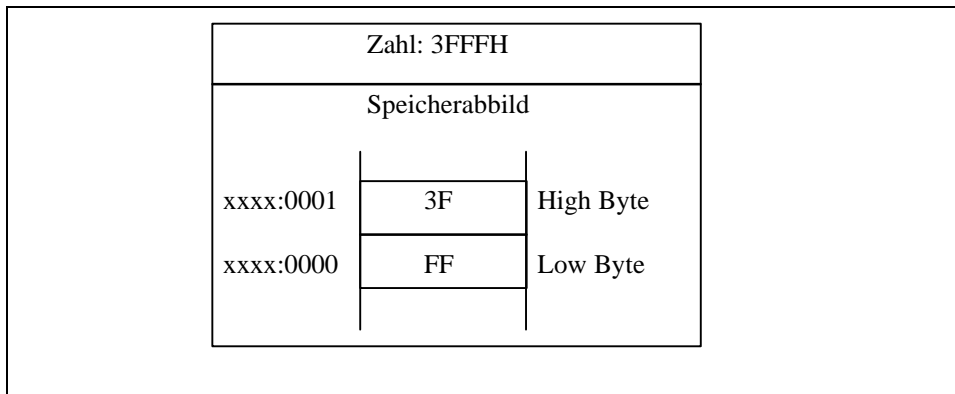


Bild 2.7: Speicherabbildung von 16-Bit-Zahlen

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).

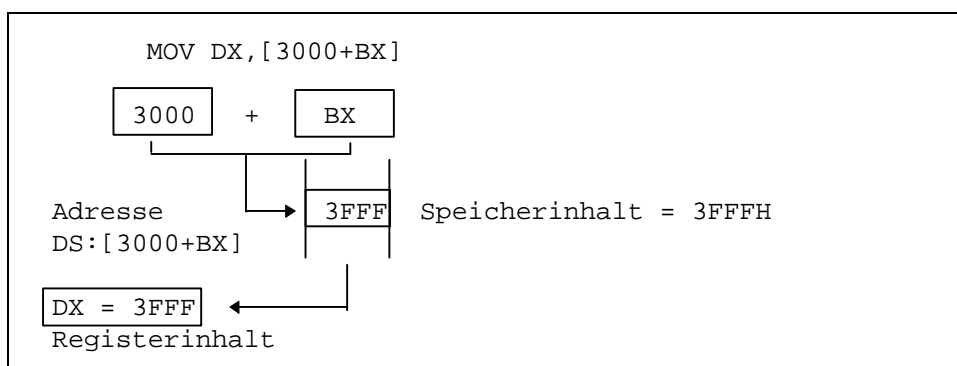


Bild 2.12: Indirekte Adressierung beim MOV-Befehl

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).

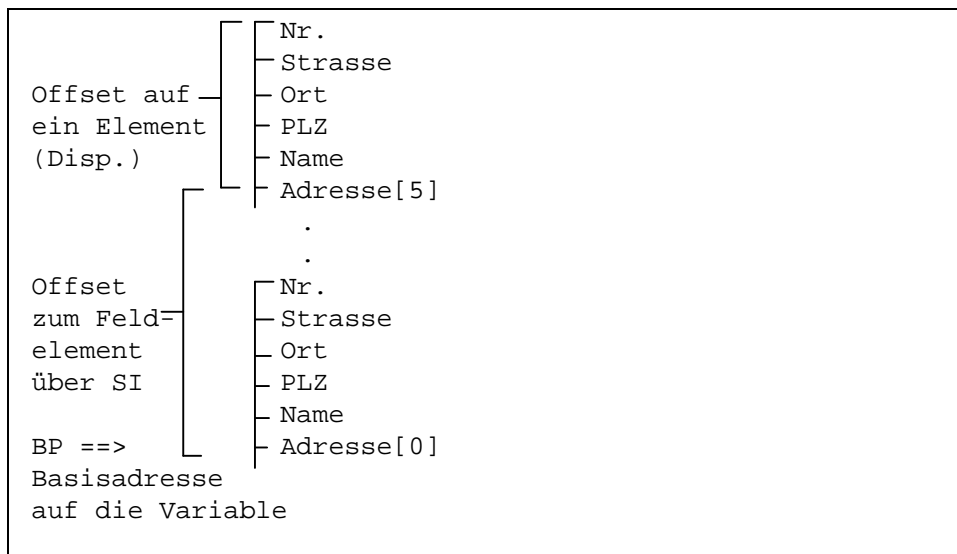


Bild 2.13: Zugriff auf eine Datenstruktur per indirekter Adressierung

Ohne diese Möglichkeit wäre jedesmal eine erhebliche Adressberechnung erforderlich. Nach der Beschreibung der verschiedenen Variationen des MOV-Befehls möchte ich die wichtigsten Ergebnisse nochmals zusammenfassen.

- ◆ Der MOV-Befehl kopiert einen 8- oder 16-Bit-Wert von einer Quelle zu einem angegebenen Ziel.
- ◆ Als Quelle lassen sich Register, Konstante und Speicherzellen angeben, während das Ziel auf Register und Speicherzellen beschränkt bleibt.
- ◆ Der Befehl verändert keine der 8086-Flags. Das Flagregister läßt sich im übrigen mit dem MOV-Befehl nicht ansprechen.
- ◆ Die Zahl der kopierten Bytes richtet sich nach dem Befehlstyp. Bei 8-Bit-Registern wird ein Byte kopiert, während bei 16-Bit-Registern ein Wort kopiert wird.
- ◆ Bezieht sich ein Befehl auf den Speicher und ist die Zahl der zu kopierenden Bytes nicht klar, muß der Befehl die Schlüsselworte BYTE oder WORD enthalten.
- ◆ Die Segmentregister lassen sich nicht direkt mit Konstanten (immediate) laden.

Tabelle 2.7 gibt nochmals die Adressierungsarten des MOV-Befehls in geschlossener Form wieder.

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.

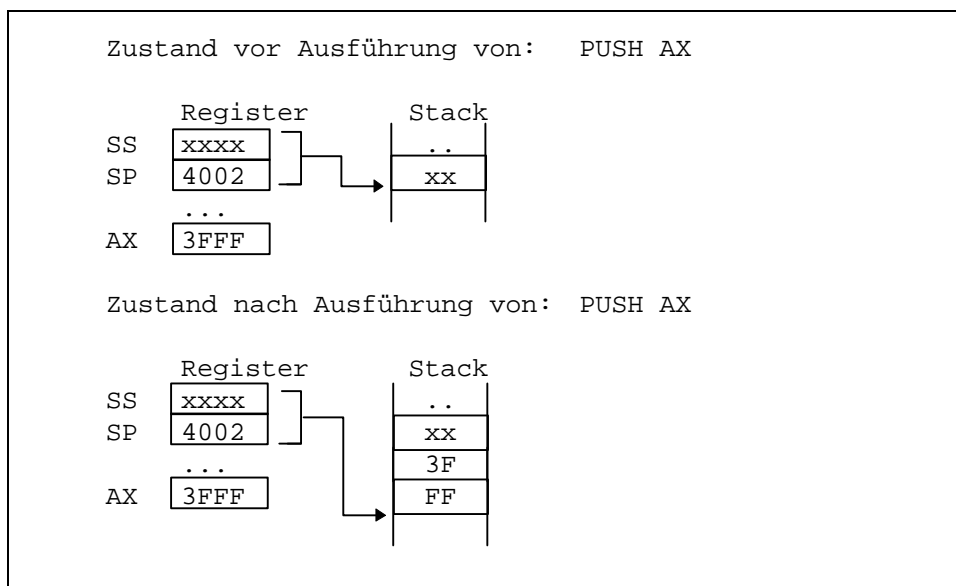


Bild 2.14: Auswirkungen des PUSH-Befehls

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 gültige 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.

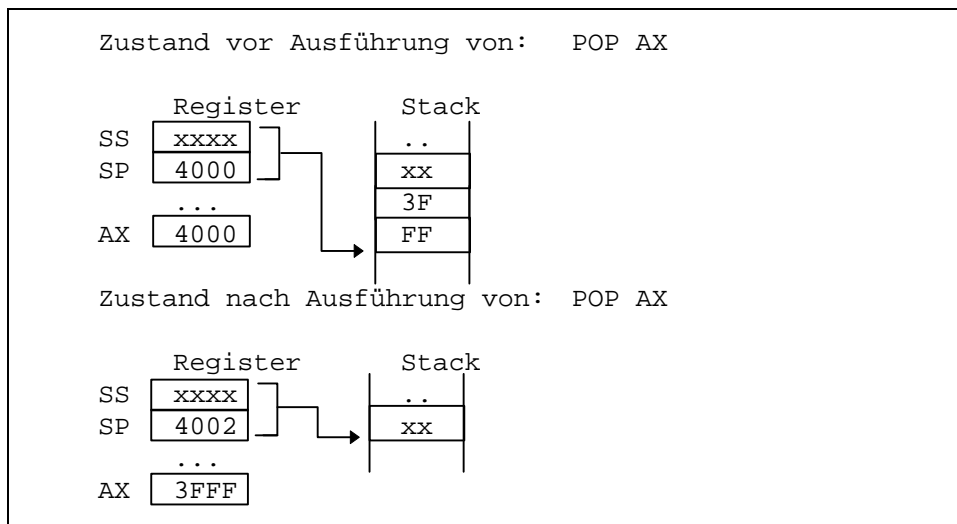


Bild 2.15: Auswirkungen des POP-Befehls

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 Beispielprogramm 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 Druckeranschluss. 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 Befehlsenerweiterung, 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 Maschinsprache 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 Ausgabe-code 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.

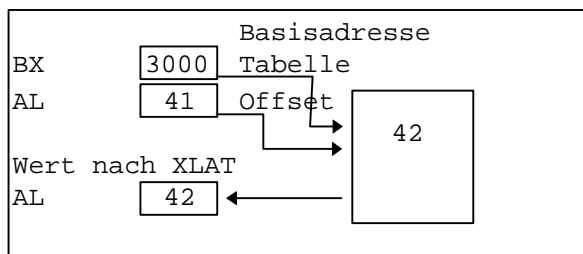


Bild 2.17: Translate Code per XLAT-Befehl

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).

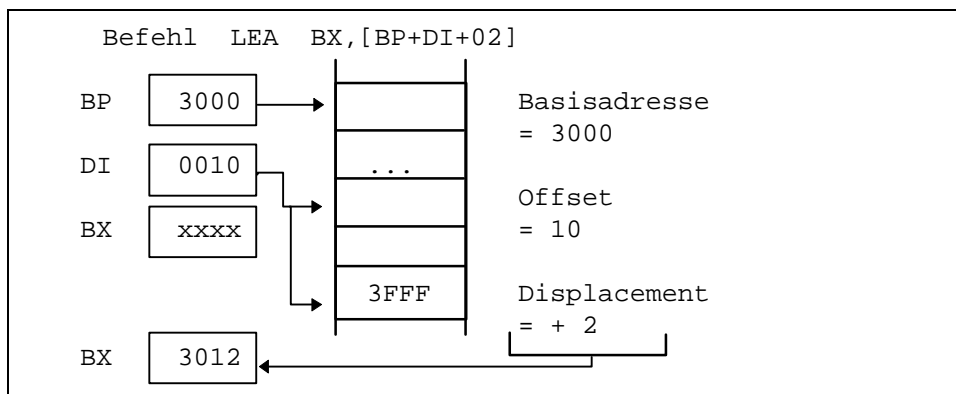


Bild 2.18: Berechnung der effektiven Adresse mit LEA

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:

$$\begin{array}{r}
 \text{BP} = 3000 \\
 \text{DI} = 0010 \\
 \quad + 02 \\
 \hline
 \text{BX} = 3012
 \end{array}$$

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:

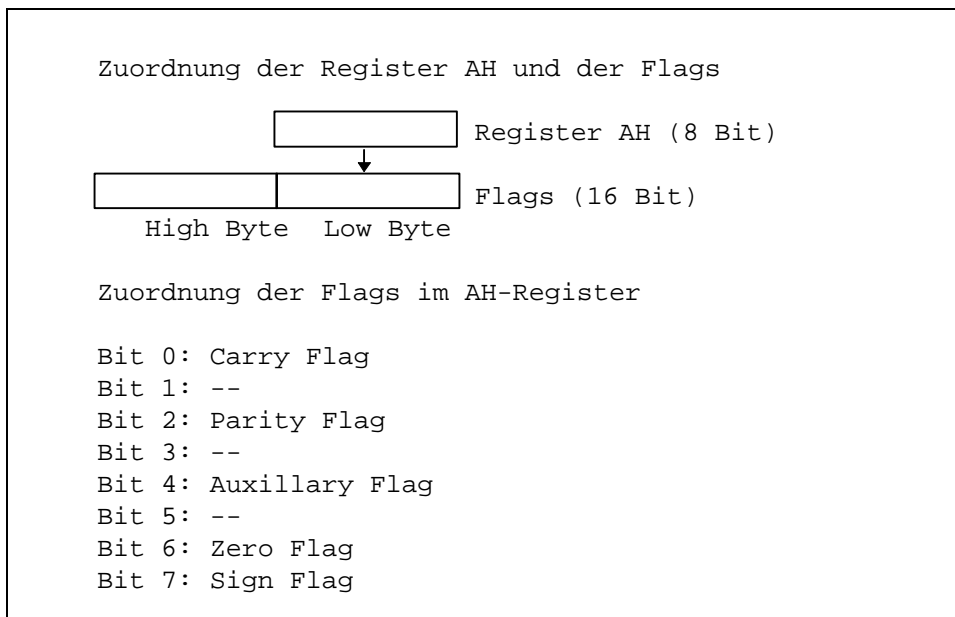
LES DI,[BP+23]

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 Zielperanden werden gemäß Bild 2.21 mit der UND-Funktion verknüpft und das Ergebnis findet sich anschließend im Zielperanden.

	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 Zielperanden 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 Zielooperanden 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 Zielooperanden 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).

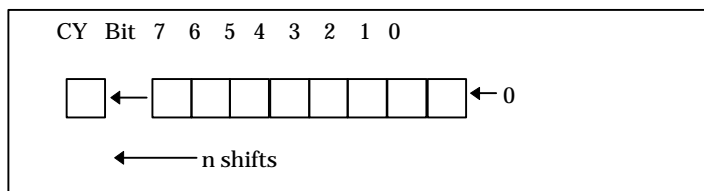


Bild 2.24 : Shift Left bei 8-Bit-Operanden

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).

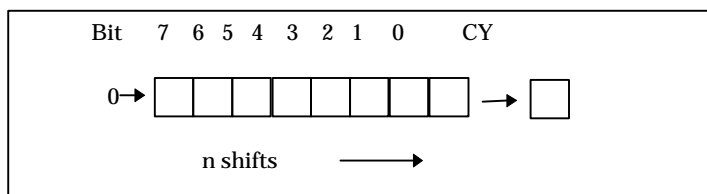


Bild 2.25: Shift Right bei 8-Bit-Operanden

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).

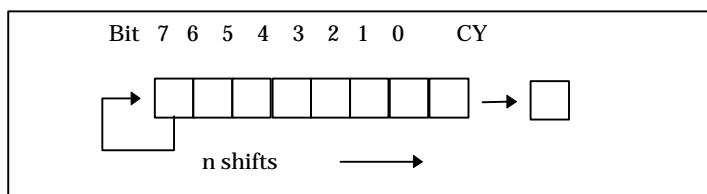


Bild 2.26 : Shift Arithmetic Right (SAR) bei 8-Bit-Operanden

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 Speichervariable benötigt der Assembler die Schlüsselworte:

```
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).

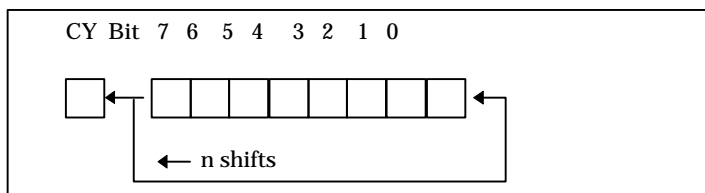


Bild 2.27: Der ROL-Befehl bei 8-Bit-Operanden

Der Befehl besitzt das Format:

ROL Ziel, Count

Count gibt dabei an, um wieviele Bitpositionen der Zielooperand 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.

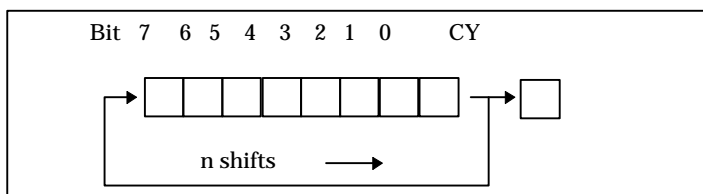


Bild 2.28: Die ROR-Operation bei 8-Bit-Operanden

## 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).

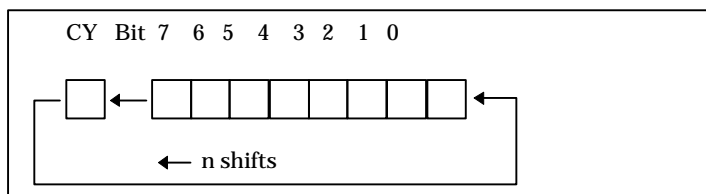


Bild 2.29: Der RCL-Befehl bei 8-Bit-Operanden

Der Befehl besitzt folgendes Format:

RCL Ziel, Count

Mit Ziel wird dabei der Zieloperand (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, daß 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).

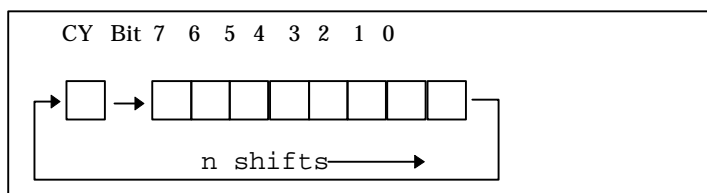


Bild 2.30: Der RCR-Befehl bei 8-Bit-Operanden

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



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).

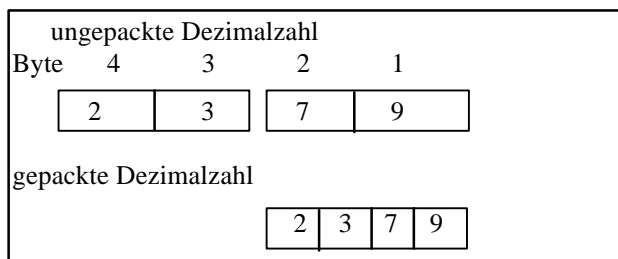


Bild 2.33: Darstellung von gepackten und ungepackten Dezimalzahlen

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.

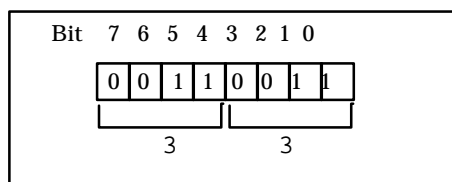


Bild 2.34: Darstellung einer BCD-Zahl

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 Zieloperanden gespeichert. Der Befehl verändert die Flags:

AF, CF, OF, PF, SF, ZF

Als Operanden dürfen Speichervariable, 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.

Table 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 Subtraktion) 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 Speichervariable 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).

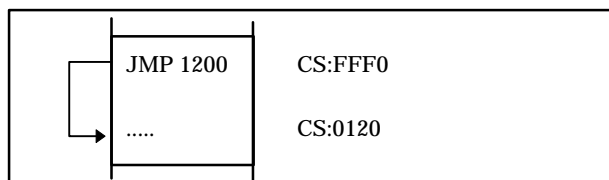


Bild 2.36: Adressüberlauf beim JMP-Befehl

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?

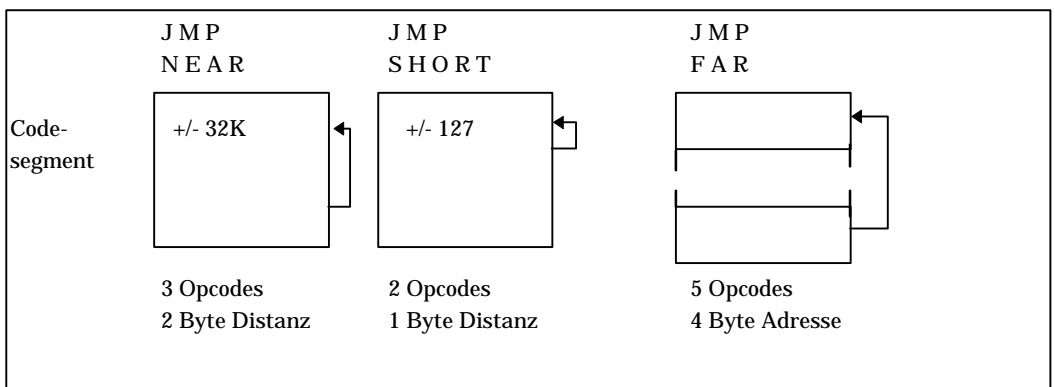


Bild 2.37: Aufstellung der verschiedenen `JMP`-Befehle

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 abzurufenden 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 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 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 XOR OF) OR ZF) = 0	$X > Y$
JGE/ JNL	Jump if Greater or Equal Jump if Not Less	(SF XOR OF)=0	$X \geq Y$
JL/ JNGE	Jump if Less Jump if Not Greater nor Equal	(SF XOR OF)=1	$X < Y$
JLE/ JNG	Jump if Less or Equal Jump if Not Greater	((SF XOR OF) 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	$X \diamond Y$ $X \diamond 0$
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 decremementiert 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
I  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 Programmstart 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:

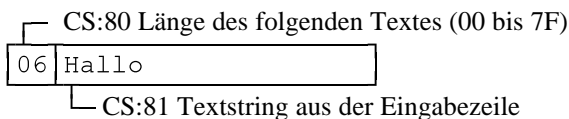


Bild 2.38: Aufbau des Transferpuffers im PSP-Bereich

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 Fehleroutine 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).

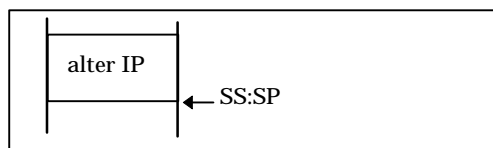


Bild 2.39: Stackzustand beim CALL NEAR-Befehl.

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 Ausgaberoutine. 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.

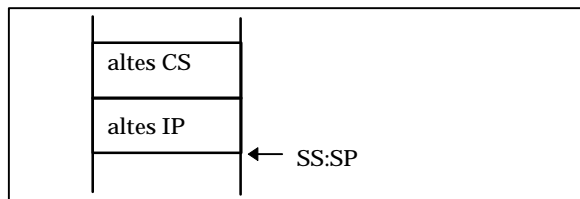


Bild 2.40: Stack beim Aufruf des CALL FAR-Befehls

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

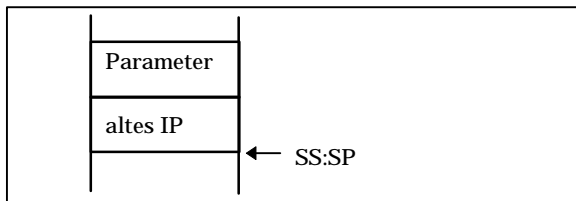
RETF ; für CALL FAR

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.

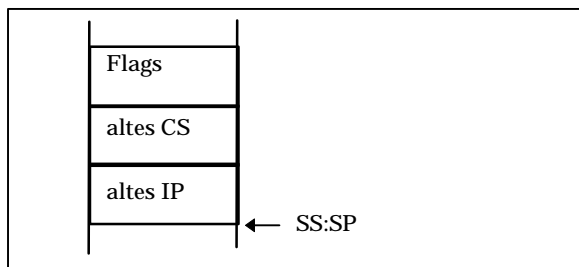


Bild 2.42: Stackzustand beim INT-Befehl

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:

INTO  
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 decremientiert (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 decremientiert. Die Verzweigung erfolgt, falls die Bedingung:

$$CX \neq 0 \quad \text{und} \quad \text{Zero Flag} = 1$$

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:

$$CX \neq 0 \quad \text{und} \quad \text{Zero Flag} = 0$$

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 plazieren. 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 Stringelements. 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.



## 3 Einführung in den A86

In Kapitel 2 wurden die Befehle des 8086-/8088-Prozessors vorgestellt. Zur Bearbeitung der Befehle wird der DOS-Debugger (DEBUG.COM) benötigt. Dadurch müssen Sie sich nicht mit den Besonderheiten der verschiedenen Assembler befassen. Zur Einführung in die Assemblerprogrammierung ist dies eine ideale Sache.

Allerdings besitzt die Assemblerprogrammierung mit DEBUG auch einige Nachteile:

- ◆ Die Assemblerprogramme lassen sich nicht oder nur mit Aufwand in andere Programme (z.B. C oder Pascal) einbinden.
- ◆ Sie können nur COM-Programme erzeugen.
- ◆ Bei Schleifen und Verzweigungen müssen Sie die Adressen der Sprungziele selbst ermitteln.

Als Alternative bietet sich der Sharewareassembler A86 an. Es handelt sich um ein sehr leistungsfähiges Programm, das Sie ausgiebig testen können. Damit Sie direkt mit dem A86 in die Assemblerprogrammierung einsteigen können, finden Sie das Programm auf der CD-ROM. Die Dokumentation zum A86 ist ebenfalls in Textdateien auf der CD-ROM gespeichert. Wer sich jedoch nicht so gut mit der englischen Sprache auskennt, findet in diesem Kapitel die gleiche Einführung in die Assemblerprogrammierung wie in Kapitel 2 für DEBUG.

**Anmerkungen:** Wenn Sie aber mit dem A86 programmieren und das Programm häufiger benutzen, müssen Sie sich bei Eric Isaacson registrieren lassen. Für ca. \$50 erhalten Sie die neueste Dokumentation des Produkts. Dies erlaubt dem Autor des Sharewareprogramms die Weiterentwicklung des A86/D86. Weitere Hinweise bezüglich der Sharewareregistrierung und der Weitergabe von A86-Programmen finden Sie in der Dokumentation des Produkts.

Beachten Sie auch, daß das Produkt kostenlos auf der CD-ROM beige packt wurde. Autor und Verlag übernehmen weder eine juristische noch sonst eine Verantwortung oder Haftung für dieses Produkt. Es kann auch keine Unterstützung bezüglich des Produkts oder anderer Programme gegeben werden. Hierzu wenden Sie sich bitte direkt an den Hersteller.

## 3.1 Einführung in die 8086-Architektur

Die Befehle der 80x86-CPU's sind untereinander kompatibel. Selbst mit dem Pentium-Chip lassen sich die 8086-Befehle ausführen. Sobald Sie die CPU unter DOS betreiben, läuft der Prozessor im 8086-kompatiblen Modus. Dieser Modus wird auch als *Real Mode* bezeichnet. 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.

**Anmerkung:** Sie können auch in Windows 3.x eine DOS-Box starten und dort die Assemblerprogramme entwickeln und testen. Dies hat sogar den Vorteil, daß Fehler im Assemblerprogramm nicht zu einem Systemneustart führen. Sie können vielmehr die DOS-Box beenden und erneut starten. Allerdings überwacht Windows die Zugriffe auf die Speicherstellen. Der Versuch, auf eine DOS-Speicherstelle zu schreiben, wird dann mit einer Fehlermeldung abgebrochen. Windows meldet diesen Schreibversuch und bricht die Anwendung ab. Dies müssen Sie beachten, wenn Sie bestimmte Programme erstellen, die in die DOS- und BIOS-Datenbereiche schreiben möchten. Persönlich nutze ich mittlerweile Windows dazu, Assemblerprogramme zu testen.

Als Programmierer sehen Sie 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 CPU's gilt die in Bild 3.1 gezeigte Registerstruktur.

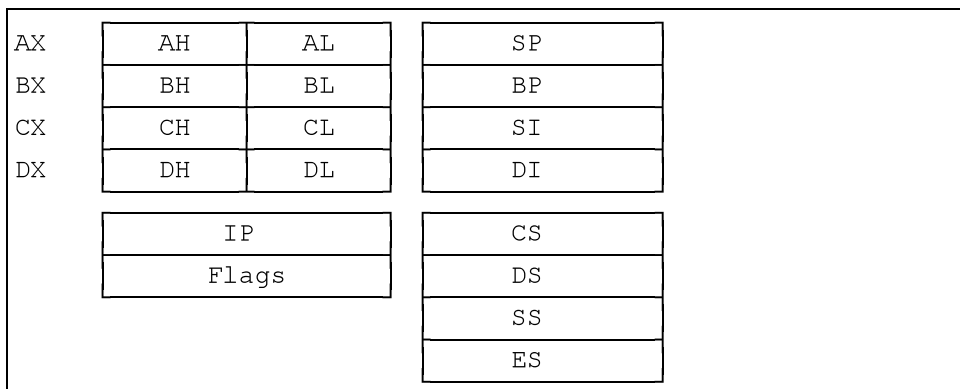


Bild 3.1: Die 8086-Registerstruktur

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.

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

### 3.1.1 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.

### **3.1.2 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.

### 3.1.3 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 3.2 ist die Kodierung dieses Registers dokumentiert.

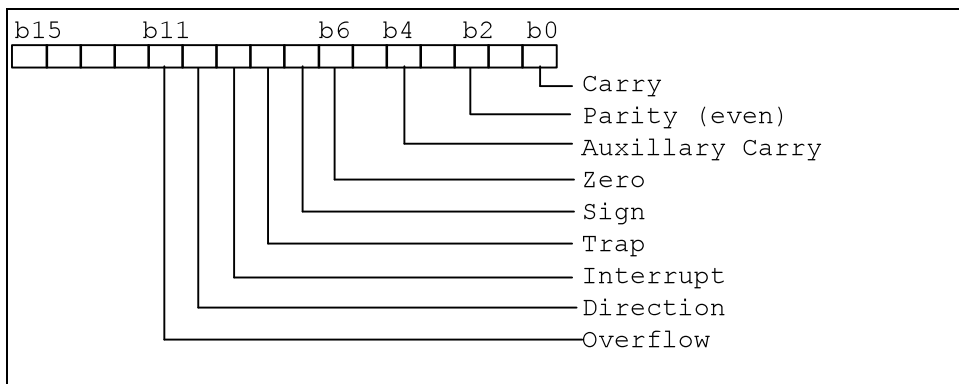


Bild 3.2: Die Kodierung der 8086-Flags

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 Mode (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 abzarbeiten.

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

### 3.1.4 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 3.3 berechnet.

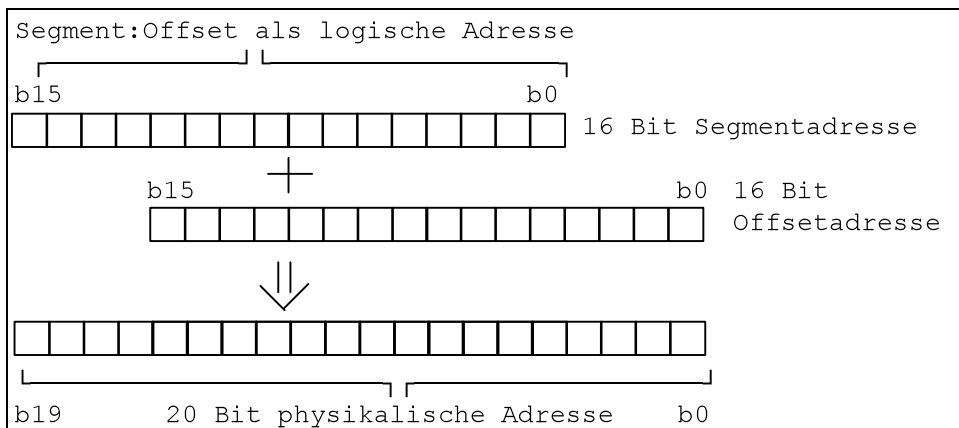


Bild 3.3: Adreßdarstellung in der Segment:Offset-Notation

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. 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 Segmentes läßt sich dann als Abstand (Offset) zum Segmentbeginn angeben (Bild 3.4).

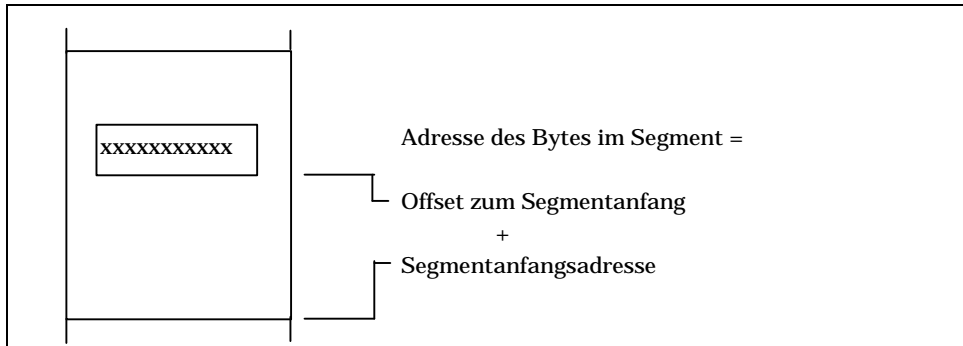


Bild 3.4: Angabe einer Speicheradresse mit Segment:Offset

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 3.3 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 3.1 gibt einige Umrechnungsbeispiele an.

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

Tabelle 3.1: Umrechnung logische in physikalische Adressen.

**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 3.2: 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.

### **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 3.5).

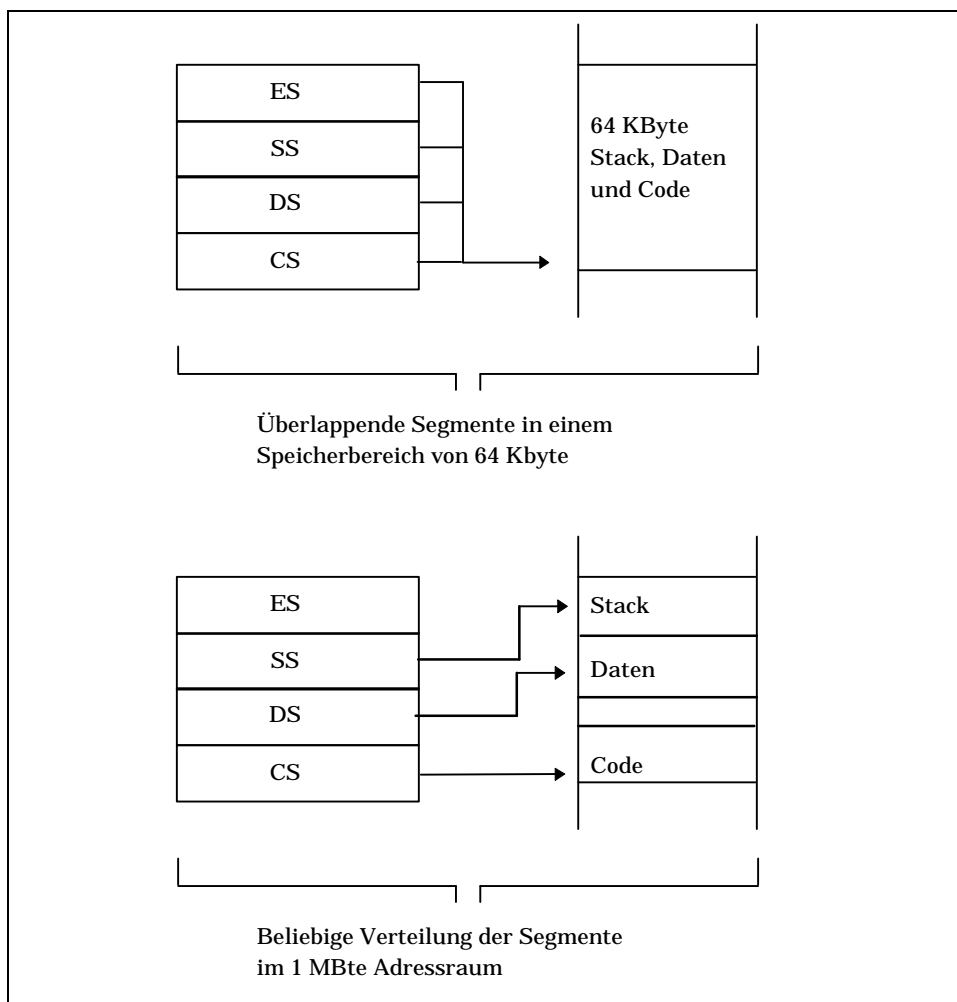


Bild 3.5: Verschiebung der Segmente im Adressraum

Die Segmentregister müssen damit erst zur Laufzeit gesetzt werden. Bei der Speicherverwaltung durch ein Betriebssystem ist dies recht vorteilhaft. So legt 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. Als Programmierer müssen Sie sich um die Belegung der Segmentregister nicht kümmern. Das Betriebssystem stellt die betreffenden Startwerte für die Segmentregister automatisch ein.

**Anmerkung:** Im 1 MByte-Adreßraum gibt es zwei Adreßbereiche, an denen keine Programme stehen sollten. Die Entwickler der 8086 CPU haben im Speicher zwei Bereiche für andere Aufgaben reserviert (Bild 3.6). 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.

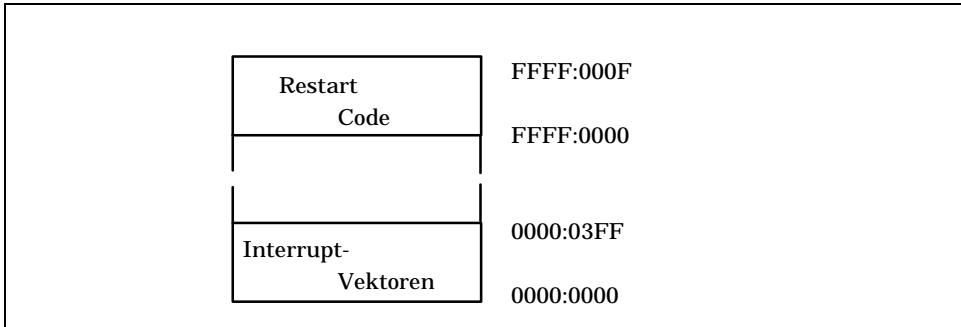


Bild 3.6: Reservierte Adreßbereiche

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. 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.

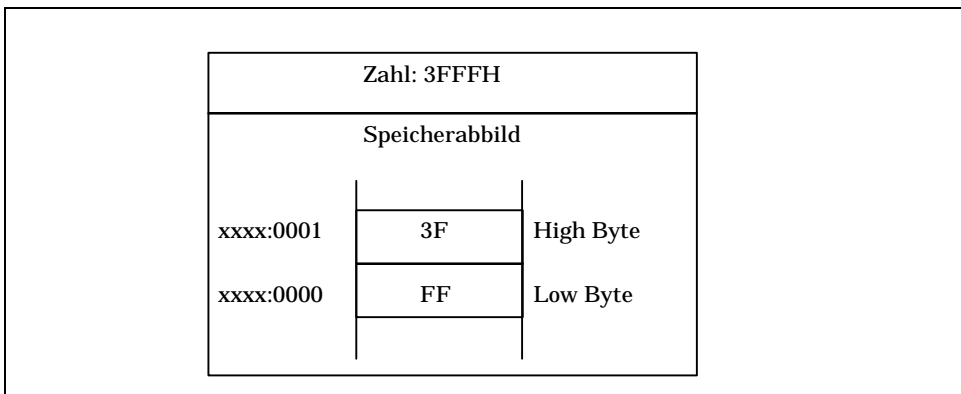


Bild 3.7: Speicherabbildung von 16-Bit-Zahlen

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) Speicherzellen 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 3.7).

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.

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 3.3 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.
String Bearbeitungs Befehle:	REP, MOVS, CMPS, LODS, STOS, etc.
Programm Transfer Befehle:	CALL, RET, JMP, LOOP, INT, IRET, etc.
Prozessor Kontroll Befehle:	NOP, STC, CLI, HLT, WAIT, etc.

Tabelle 3.3: Gruppierung der 8086-Befehle

In den nachfolgenden Abschnitten werden diese Befehle detailliert vorgestellt.

## 3.2 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.

### 3.2.1 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 3.4).

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 3.4: 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 3.8) seien die Register mit folgenden Daten vorbelegt.

AX	003F
BX	3FFF
CX	1234

Bild 3.8: Registerinhalt vor Ausführung des Befehls

Nun führt die CPU folgende Befehle aus:

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

Danach ergibt sich in den Registern die Belegung gemäß Bild 3.9.

AX	3F3F
BX	1234
CX	1234

Bild 3.9: Registerinhalt nach der Ausführung der Befehle

**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 die Einschränkung, daß die Anweisungen:

```
MOV CS,AX  
MOV IP,AX  
MOV AH,AX
```

Mit dem Befehl *MOV CS,AX* würde das Codesegmentregister mit dem Inhalt von AX überschrieben. Dies führte 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. Die meisten Assembler sperren deshalb diesen Befehl. Sie können den Seiteneffekt aber mit dem DOS-Debugger nachvollziehen (siehe Kapitel 2). Weiterhin sind Befehlskombinationen unter Verwendung des Registers IP als Operand (z.B. *MOV IP,AX*), oder die gemischte Verwendung von 8- und 16-Bit-Registern (z.B. *MOV AX,BL*) unzulässig.

## Der Immediate-MOV-Befehl

Um die Register mit gezielten Werten zu besetzen, findet der Immediate-MOV-Befehl Verwendung. Als Quelle wird dann eine Konstante direkt (immediate) aus dem Speicher gelesen und in das Ziel (Register oder Speicherstelle) kopiert. 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,03F
MOV BYTE [03000],03F
MOV WORD [04000],01234
MOV BP,01400
```

Der Wert der Konstanten darf den vorgegebenen Bereich nicht überschreiten. Werden bei einer 16-Bit-Konstanten weniger als vier Ziffern eingegeben (MOV AX,01), ersetzt der A86 die führenden Stellen durch Nullen. Bei Zuweisungen an Speicherstellen (z.B. MOV BYTE [03000],03F) überschreibt der Befehl 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 (Offset) in eckige Klammern [] gesetzt ist. Mit den Schlüsselwörtern BYTE oder WORD wird die Breite der zu speichernden Konstanten definiert. Der Immediate-MOV-Befehl kann auf die Register:

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

angewandt werden.

AX	3F3F
BX	1234
CX	1234

Bild 3.10: Registerinhalt vor den Immediate-MOV-Befehlen

**Achtung:** Direkte Zuweisungen von Konstanten an Segmentregister wie MOV DS,3500 sind unzulässig.

Beispiel

Die Wirkung eines Immediate MOV-Befehls lässt sich am besten an einem kleinen Beispiel erläutern. Ein Register soll mit einem Wert geladen werden. Bild 3.10 gibt die Registerbelegung vor der Ausführung der Immediate-MOV-Befehle an.

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 3.11.

AX	0000
BX	FF34
CX	1234

Bild 3.11: Registerinhalt nach den Immediate-MOV-Befehlen

Programmbeispiel

Damit kommen wir zu unserem ersten kleinen Programmbeispiel. Es wird der Text:

Hallo A86

auf dem Bildschirm ausgegeben. Listing 3.1 gibt die Assembleranweisungen wieder. Die Datei HALLO.ASM finden Sie auf der CD-ROM. Übersetzen Sie diese mit dem A86 mit der Anweisung:

A86 HALLO.ASM

in eine COM-Datei. Eine Auflistung der Übersetzeranweisungen finden Sie im Kapitel über die Assemblerdirectiven. Bei Fehlern meldet der A86 diese und schreibt die Meldungen in eine ERR-Datei.

```

;=====
; File: HALLO.ASM
; Programm zur Ausgabe der Meldung:
; "HALLO A86" auf dem Bildschirm.
; Programm als COM-Datei übersetzen!!
;=====
;
RADIX 16 ; Hexadezimalsystem
```

```

        ORG 0100          ; Startadresse COM
        CODE SEGMENT
;
HALLO:  MOV AH,09        ; DOS-Display-Code
        MOV DX,OFFSET TEXT ; Textadresse
        INT 21          ; DOS-Ausgabe
;
        MOV AX,4C00     ; DOS-Exit-Code
        INT 21          ; terminiere
;
; Textstring
;
TEXT:   DB 'Hallo A86',0D,0A,'$'
;
END Hallo

```

*Listing 3.1: Das Programm HALLO.ASM*

Alle Anweisungen hinter einem Semikolon bis zum Zeilenende werden als Kommentar interpretiert. Die ersten drei Anweisungen dienen zur Steuerung des Assemblers. In der ersten Anweisung wird zum Beispiel vereinbart, daß alle Zahlen im Hexadezimalsystem angegeben werden. Näheres hierzu findet sich im Kapitel über die Assemblerdirectiven. Mit ORG 100 wird z.B. der Code für eine COM-Datei vorbereitet, die immer ab CS:0100H beginnen muß. Das eigentliche Programm ist recht einfach:

- ◆ Mittels des DOS-Aufrufes INT 21, AH = 09 wird der Text auf dem Bildschirm ausgegeben.
- ◆ Die Textadresse ist in den Registern DS:DX zu übergeben. Da in COM-Dateien CS = DS = ES = SS ist, enthält DS bereits den korrekten Wert.
- ◆ Der Offset wird durch MOV DX, OFFSET TEXT geladen. Der Text wird mit der DB-Anweisung definiert - wichtig ist dabei, den String mit dem \$-Zeichen abzuschließen.
- ◆ Assemblerprogramme werden mit dem DOS-Aufruf INT 21, AH = 4CH beendet.

Nähere Hinweise über die Interna von DOS und zur Belegung der DOS-INT 21-Aufrufe finden sich in /1/ (siehe Literaturhinweise).

## Indirekte Adressierung beim MOV-Befehl

Oft möchte der Programmierer ein Ergebnis aus einem Register auf eine Speicherstelle zurückspeichern. Mit der indirekten Adressierung läßt sich dies leicht durchführen. 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 Adresse ist dabei in eckigen Klammern anzugeben, wobei der Ausdruck aus Konstanten und den Registern BX, BP, SI und DI kombiniert werden darf. Die Adresse der Speicherstelle wird in obigem Beispiel indirekt über ein Register [SI] oder über einen Ausdruck [3000+BX] angegeben. Der Prozessor berechnet sich die Adresse aus dem Ausdruck in den eckigen Klammern [].

Die letzte Anweisung liest den Inhalt der Speicherstelle DS:[BX+3000] und speichert das Wort in das Register DX. Demnach wird 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 3.12).

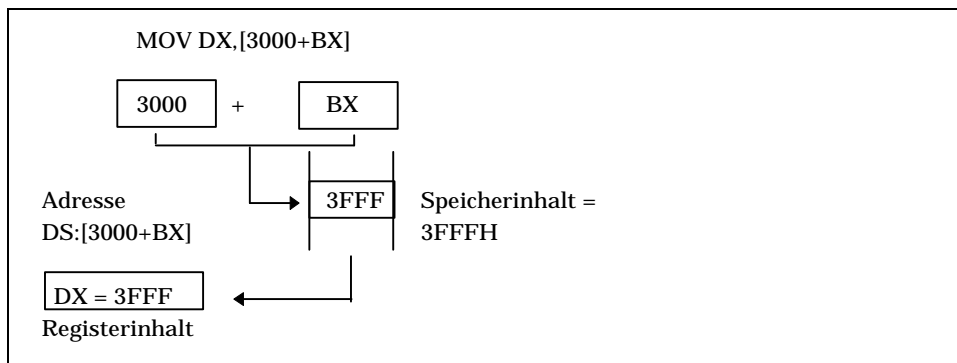


Bild 3.12: Indirekte Adressierung beim MOV-Befehl

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 3.5: Indirekte Adressierung über Register

In Bild 3.12 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 Ausführung der Anweisung enthält das Register DX dann den Inhalt des Wortes an der Adresse DS:[3000+BX].

Tabelle 3.5 gibt die Möglichkeiten zur Berechnung der indirekten Adressen an. 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 Zielerand dürfen beim indirekten MOV-Befehl 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). Damit sind zum Beispiel folgende Befehle zulässig:

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

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 Stacksegmentes. Es wird also das SS-Register zur Ermittlung des Segmentes benutzt. Bei Verwendung der indirekten Adressierung gilt daher die Zuordnung:

```
BP -> SS-Register
BX -> DS-Register
```

Dieser Sachverhalt ist bei der Programmerstellung zu beachten.

### Programmbeispiel

Damit kommen wir zu einem weiteren 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 Textmode 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. Die Quelldatei mit dem Namen:

SCREEN.ASM

übersetzen Sie mit der folgenden Anweisung:

A86 SCREEN.ASM

Sofern keine Fehlermeldungen auftreten, existiert anschließend auf dem Standardlaufwerk ein ablauffähiges COM-Programm mit dem Namen SCREEN.COM.

```
=====
; File: SCREEN.ASM   (c) Born G.
; Funktion: A86-Programm zur Ausgabe
; auf den Bildschirm. Aufruf:
;
;   SCREEN
;
=====
;
;   RADIX 16           ; Hexadezimalsystem
;   ORG 0100          ; Startadresse COM
;   CODE SEGMENT
;
SCREEN:
;   MOV AX,0B800      ; Adresse Bildschirm
;   MOV DS,AX         ; Segment setzen
;   MOV BYTE [0500],41 ; 'A'
;   MOV BYTE [0501],7F ; Attribut
;
;   ; DOS-Exit, Returncode steht bereits in AL
;
;   MOV AH,4C         ; INT 21-Exitcode
;   INT 21            ; terminiere
;
END Screen
```

*Listing 3.2: Screenausgabe*

Nachdem Sie das Programm mit A86 übersetzt haben, laden Sie die COM-Datei mit SCREEN.COM

Mit dem ersten Assemblerbefehl (MOV AX,0B800) wird die Segmentadresse des Bildschirmspeichers definiert. Diese liegt bei B000 oder B800 und ist in das DS-Register zu kopieren.

**Anmerkung:** Immer wenn Sie eine Hexadezimalzahl verwenden, müssen Sie auf die erste Ziffer achtgeben. Entspricht die Hexziffer einem Buchstaben (A, B, C, D, E, F),

muß eine 0 vorangestellt werden. Andernfalls interpretiert der A86 diese Ziffer als den ersten Buchstaben einer Variablen. Dies führt dann zu einer Fehlermeldung, wobei die Ursache nur schwer zu erkennen ist. Die Segmentadresse B800 muß deshalb als 0B800 angegeben werden.

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 Segmentes 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.

## 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 Befehle 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.

**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 Stacksegmentes. Es wird also das SS-Register zur Ermittlung des Segmentes 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 3.6 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 3.6: Segmentregister bei der indirekten Adressierung

DISP steht hier für eine Konstante.

**Anmerkung:** 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;
```

Elemente solcher Strukturen lassen sich sehr einfach durch die indirekte Adressierung ansprechen. Es muß nur 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. In Kapitel 2 wird gezeigt, wie die Assembleranweisungen zum Zugriff auf solche Datenstrukturen aussehen.

Nach der Beschreibung der verschiedenen Variationen des MOV-Befehls möchte ich die wichtigsten Ergebnisse nochmals zusammenfassen.

- ◆ Der MOV-Befehl kopiert einen 8- oder 16-Bit-Wert von einer Quelle zu einem angegebenen Ziel.
- ◆ Als Quelle lassen sich Register, Konstante und Speicherzellen angeben, während das Ziel auf Register und Speicherzellen beschränkt bleibt.
- ◆ Der Befehl verändert keine der 8086-Flags. Das Flagregister läßt sich im übrigen mit dem MOV-Befehl nicht ansprechen.
- ◆ Die Zahl der kopierten Bytes richtet sich nach dem Befehlstyp. Bei 8-Bit-Registern wird ein Byte kopiert, während bei 16-Bit-Registern ein Wort kopiert wird.
- ◆ Bezieht sich ein Befehl auf den Speicher und ist die Zahl der zu kopierenden Bytes nicht klar, muß der Befehl die Schlüsselworte BYTE oder WORD enthalten.

- ◆ Die Segmentregister lassen sich nicht direkt mit Konstanten (immediate) laden.

Tabelle 3.7 gibt nochmals die Adressierungsarten des MOV-Befehls in geschlossener Form wieder.

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 3.7: Adressierungsarten des MOV-Befehls (Ende)

**Anmerkung:** Manchmal finden Sie noch spezielle Bezeichnungen für die einzelnen Adressierungsarten des MOV-Befehls (z.B. Basis-Adressierung, Index-Adressierung etc.). In Kapitel 2 werden diese Begriffe im Abschnitt "Die Adressierungsarten für Speicherzugriffe" erläutert.

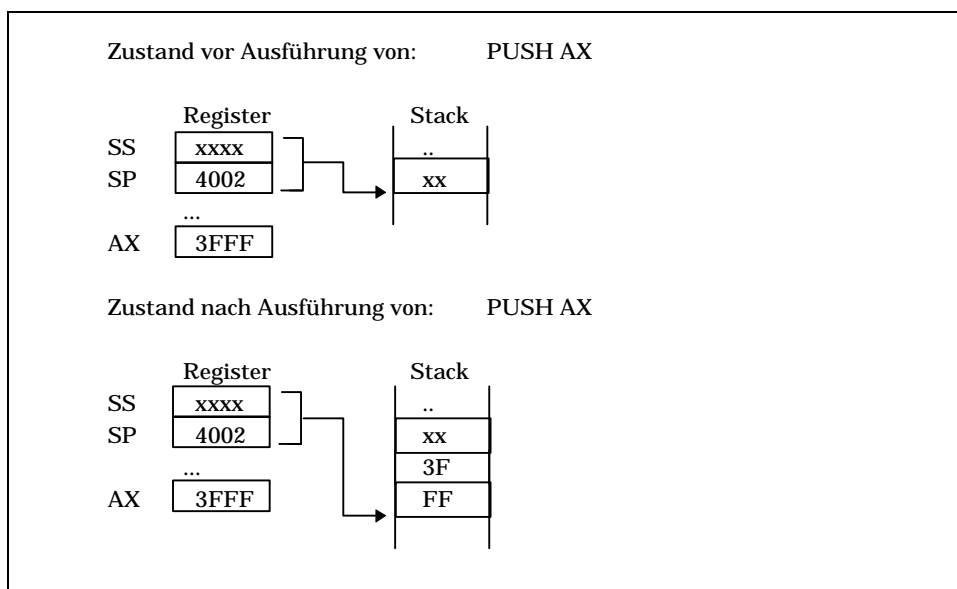


Bild 3.14: Auswirkungen des PUSH-Befehls

### 3.2.2 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 3.14 zeigt den Ablauf beim PUSH-Befehl.

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 3.8 gibt einige gültig PUSH-Anweisungen wieder.

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

Tabelle 3.8: 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 gültige 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 Daten-segment (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.

**Anmerkung:** Bei Verwendung des 80286-Befehlssatzes (und des NEC V20) gibt es noch die Befehle:

```
PUSH imm8
PUSH imm16
```

die eine 8- oder 16-Bit-Konstante auf dem Stack ablegen. Mit der Anweisung:

```
PUSHA
```

lassen sich beim 80286/80386 die Register AX, CX, DX, BX, SP, BP, SI und DI mit einmal auf dem Stack sichern.

### **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? 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.

### 3.2.3 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 3.15 verdeutlicht die Arbeitsweise des POP-Befehls.

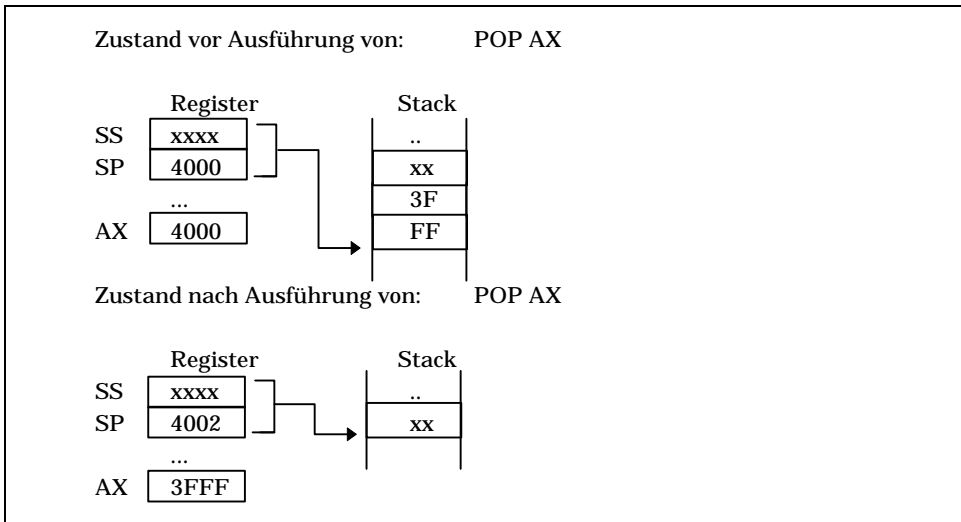


Bild 3.15: Auswirkungen des POP-Befehls

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. Tabelle 3.9 enthält eine Aufstellung möglicher POP-Befehle.

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

Tabelle 3.9: 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.

**Achtung:** Der POP-Befehl darf nicht auf die Register CS, SS und SP angewandt werden. 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
```

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 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. Auch wenn es die entsprechenden Maschinenbefehle gibt, ist es 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.

## 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.

### Programmbeispiel 1

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
```

Dieses Programm ist nicht als Listing auf der CD-ROM gespeichert. Bei Bedarf lassen sich die Anweisungen aber in eigene Programme einbinden.

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 2

Nach diesen Ausführungen möchte ich ein weiteres 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 Druckerkabel an den Anschlußports getauscht werden, eine umständliche und nicht ganz befriedigende Möglichkeit.

Hier setzt unser Beispielprogramm 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 3.16: 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. Das Programm ist nicht auf der CD-ROM gespeichert, da später eine erweiterte Version vorgestellt wird. Sie können die Anweisungen jedoch leicht mit einem Texteditor eingeben und in der Datei:

### SWAP1.ASM

speichern.

```

;=====
; File: SWAP1.ASM   (c) Born G.
; Programm zur Vertauschung von LPT1 und LPT2.
; Programm als COM-Dateiübersetzen!!
;=====
;
        RADIX 16           ; Hexadezimalsystem
        ORG 0100          ; Startadresse COM
        CODE SEGMENT
;
SWAP:  MOV AX,0000        ; ES auf Segm. 0000
        MOV ES,AX        ; setzen
        ES:MOV AX,[0408] ; Portadresse LPT1
        PUSH AX          ; merke Wert auf Stack
        ES:MOV AX,[040A] ; Portadresse LPT2
        ES:MOV [0408],AX ; store Portadressen
        POP AX

```

```

        ES:MOV [040A],AX      ;
;
;
        MOV AX,4C00          ; DOS-Exit Code
        INT 21              ; terminiere
END Swap

```

*Listing 3.3: SWAP-Programm (Version 1)*

Die Quelldatei SWAP1.ASM läßt sich mit:

A86 SWAP1.ASM

übersetzen. Falls keine Fehler auftreten liegt eine ausführbare COM-Datei vor. Sie können die Wirkungsweise des Programmes mit DEBUG austesten (DEBUG SWAP1.COM). Wird das Programm zum ersten Mal ausgeführt, vertauscht es die Belegung der Druckerports. 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 Drucker-gänge 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.

Am Ende des Assemblerprogramms einige Anweisungen, die DOS mitteilen, daß das Programm beendet werden soll. Programme können mit DOS über eine Art Programm-bibliothek 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 SWAP.ASM lernen Sie in den folgenden Abschnitten kennen.

### 3.2.4 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 Portadresse im Bereich zwischen 00H und FFH angegeben. Der Befehl liest nun einen 8/16-Bit-Wert aus dem angegebenen Port aus und speichert das Ergebnis im Akkumulator. Die Registerbreite (AX oder AL) spezifiziert dabei, ob ein Wort oder Byte zu lesen ist. 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 Befehlsenerweiterung, 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.

### 3.2.5 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,0FFFF; 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.

### 3.2.6 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 3.10 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 3.10: 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. Die Datei findet sich unter dem Namen SWAP.ASM auf der CD-ROM.

```

;=====
; File: SWAP.ASM   (c) Born G.
; Programm zur Vertauschung von LPT1
; und LPT2. Programm als COM-Datei
; übersetzen!!
;=====
;
;       RADIX 16           ; Hexadezimalsystem
;       ORG 0100           ; Startadresse COM
;       CODE SEGMENT
;
SWAP:  MOV AH,09           ; DOS-Display Code
;       MOV DX,OFFSET TEXT ; Textadresse
;       INT 21             ; DOS-Ausgabe
;
;       MOV AX,0000        ; ES auf Segm. 0000
;       MOV ES,AX          ; setzen
;       ES:MOV AX,[0408]   ; Portadresse LPT1
;       ES:MOV BX,[040A]   ; Portadresse LPT2
;       XCHG AX,BX        ; Swap Adressen
;       ES:MOV [0408],AX   ; store Portadressen

```

```
ES:MOV [040A],BX      ;
;
MOV AX,4C00           ; DOS-Exit Code
INT 21               ; terminiere

;
; Textstring
;
TEXT: DB 'SWAP LPT1 <-> LPT2 (c) Born G.',0D,0A,'$'
;
END Swap
```

Listing 3.4: SWAP.ASM

Das Programm kommt nun ohne Zwischenspeicher (Memory oder Stack) aus. Ein weiterer Aufruf von SWAP stellt wieder den ursprünglichen Zustand her. Die Belegung der BIOS-Variablen ist /1/ (siehe Literaturhinweise) zu entnehmen.

**Anmerkung:** Der XCHG-Befehl läßt sich noch zur Realisierung von Semaphoren verwenden. Lesen Sie bei Bedarf in Kapitel 2 im Abschnitt *Semaphore mit XCHG* nach.

### 3.2.7 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 Maschinsprache ein Byte (Opcode 90H). Im Befehlssatz des 8086-Prozessors wurde deshalb die zusätzliche Anweisung:

```
NOP
```

vorgesehen. Der A86-Assembler generiert aber für die Anweisungen XCHG AX,AX und 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.

### 3.2.8 Der XLAT-Befehl

Zur Umcodierung von Werten läßt sich der XLAT-Befehl verwenden. 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 Befehl:

XLAT

liest den durch BX+AL adressierten Tabellenwert und gibt diesen im Register AL zurück. Wichtig ist aber, daß vor Benutzung der Anweisung das Register BX mit der Anfangsadresse der Tabelle und AL mit dem zu konvertierenden Zeichen 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. Bild 3.17 gibt die Wirkungsweise des Befehls schematisch wieder.

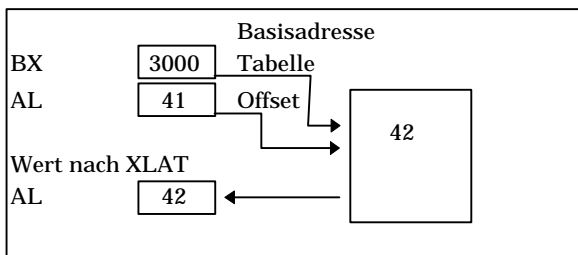


Bild 3.17: Translate Code per XLAT-Befehl

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 3.17 wird der Wert 41H aus AL durch den Tabellenwert 42H ersetzt.

**Anmerkung:** In Kapitel 2 finden Sie weitere Erläuterungen zum XLAT-Befehl

### 3.2.9 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 3.18 und speichert das Ergebnis im Zielregister (hier BX).

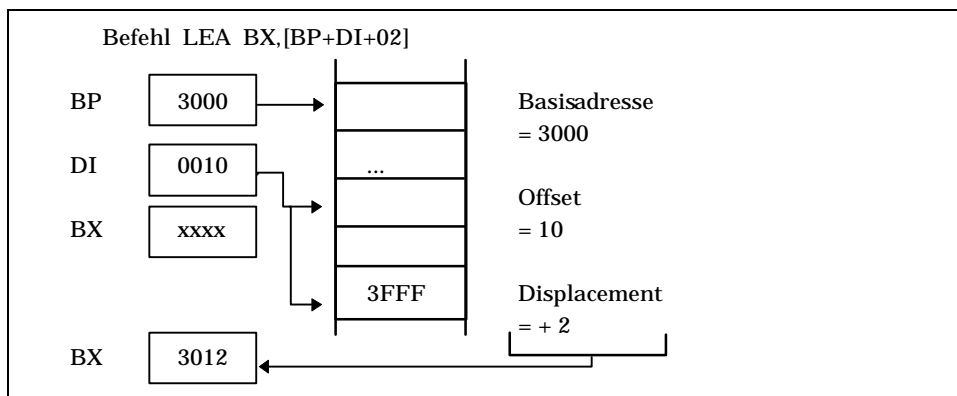


Bild 3.18: Berechnung der effektiven Adresse mit LEA

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:

$$\begin{array}{r}
 \text{BP} = 3000 \\
 \text{DI} = 0010 \\
 \hline
 \quad + \quad 02 \\
 \hline
 \text{BX} = 3012
 \end{array}$$

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.

### 3.2.10 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 Ziel-Register 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 LES-Befehl

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.

### 3.2.11 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.

#### Der Befehl LAHF

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 3.19 gezeigte Zuordnung.

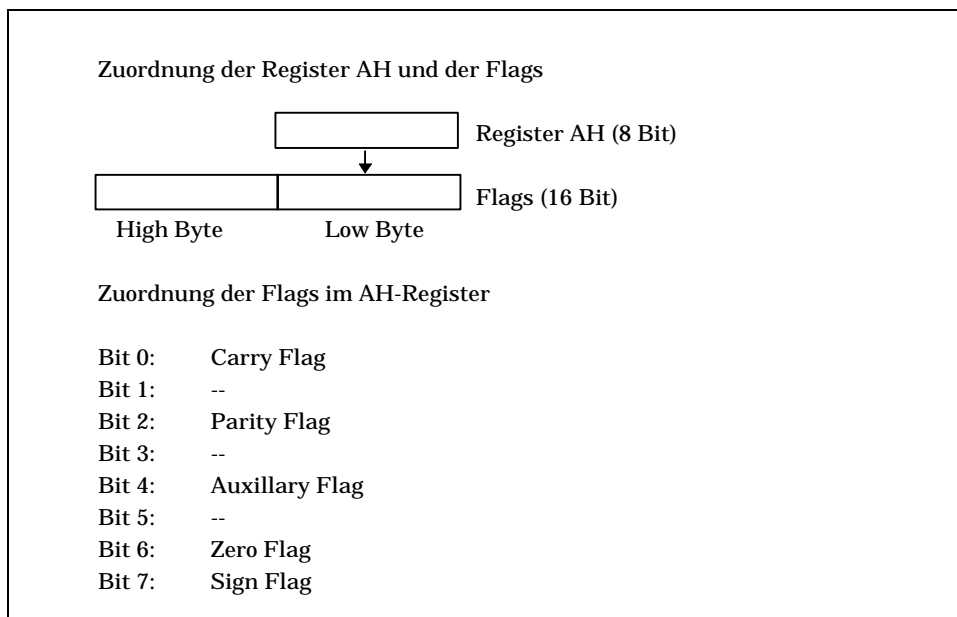


Bild 3.19: Austausch der Flags und AH über die Befehle LAHF und SAHF

Der Befehl *Store AH-Register to Flags (SAHF)* transferiert den Inhalt des AH-Registers zum 8086-Flag-Register. Es gilt dabei die in Bild 3.19 gezeigte Zuordnung. Die beiden Befehle wurden der Vollständigkeit halber aufgeführt. Sie kommen aber in den Beispielpogrammen der folgenden Kapitel nicht vor.

## 3.3 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.

### 3.3.1 Der NOT-Befehl

Mit der NOT-Anweisung werden alle Bits des Operanden invertiert (Bild 3.20).

NOT	1001 0111
=>	0110 1000

Bild 3.20: NOT-Operation

Der Befehl besitzt folgende Syntax:

NOT Operator

Als Operator lassen sich dabei Register oder Speichervariable (Byte oder Word) angeben. Tabelle 3.11 enthält eine Zusammenfassung gültiger NOT-Befehle.

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

Tabelle 3.11: Die NOT-Befehle

Bei Speicherzugriffen benötigt der Assembler die Schlüsselworte:

NOT WORD [3000]  
NOT BYTE [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.

**Anmerkung:** Bei DEBUG wird bei Speicherzugriffen das Schlüsselwort *PTR* (z.B. BYTE PTR) benutzt. Das Schlüsselwort *PTR* benötigt der A86 nicht.

### 3.3.2 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 3.21 mit der UND-Funktion verknüpft und das Ergebnis findet sich anschließend im Zieloperanden.

	0101 1100
AND	1011 1111
=>	0001 1100

Bild 3.21: AND-Operation

Tabelle 3.12 enthält eine Aufstellung gültiger AND-Befehle.

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

Tabelle 3.12: 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 [BX+3000], 3000

AND BYTE [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. Das Programm ist in der Textdatei NUMOFF.ASM auf der CD-ROM gespeichert. Übersetzen diese mit A86 in eine COM-Datei:

A86 NUMOFF.ASM

Anschließend muß eine ausführbare COM-Datei mit dem Namen:

NUMOFF.COM

vorliegen. Das Programm nutzt wieder Wissen über die Belegung des BIOS-RAM-Bereichs. In der Speicherzelle 0000:0417 speichert das BIOS wie die einzelnen Tasten (NumLock, CapsLock, etc.) gesetzt sind. Die genaue Kodierung ist in /1/ aufgeführt. Bit 5 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.

```

;=====
; File: NUMOFF.ASM (c) Born G.
; Ausschalten der NUM-Lock-Taste.
; Programm als COM-Datei übersetzen!!
;=====
;
;          RADIX 16          ; Hexadezimalsystem
;          ORG 0100          ; Startadresse COM
;          CODE SEGMENT
;
NUMOFF:MOV AH,09             ; DOS-Display Code
;          MOV DX,OFFSET TEXT ; Textadresse
;          INT 21             ; DOS-Ausgabe
;
;          MOV AX,0000        ; ES auf BIOS-Segm.
;          MOV ES,AX          ; setzen
;          ES:AND BYTE [0417],0DF ; Bit ausblenden
;
;          MOV AX,4C00        ; DOS-Exit Code
;          INT 21             ; terminiere
; Textstring
TEXT: DB 'NUMOFF (c) Born G.',0D,0A,'$'
;
END NUMOFF

```

Listing 3.5: NUMOFF.ASM (Version A86)

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

```
AND BYTE [0417],DF
```

verknüpft. Da der Assembler nicht weiß, ob ein Byte oder ein Wort zu bearbeiten ist, muß explizit die Angabe *BYTE* im Befehl auftreten. Die restlichen Befehle sind aus den vorhergehenden Beispielen bereits bekannt.

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.

### 3.3.3 Der OR-Befehl

Mit der OR-Anweisung läßt sich eine weitere logische Verknüpfung gemäß Bild 3.22 durchführen.

	0111 0000
OR	1010 1001
=>	1111 1001

Bild 3.22: 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 3.13 enthält eine Aufstellung gültiger OR-Befehle.

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

Tabelle 3.13: 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 benötigt der A86-Assembler die Schlüsselworte:

```
OR WORD [BX+3000], 3000
OR BYTE [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).

### 3.3.4 Der XOR-Befehl

Mit der XOR-Anweisung wird eine Verknüpfung gemäß Bild 3.23 durchgeführt.

	1010 1001
XOR	1001 1011
=>	0011 0010

Bild 3.23: 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 3.14 enthält eine Aufstellung gültiger XOR-Befehle.

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

Tabelle 3.14: 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 Speichervariable benötigen die Assembler die Schlüsselworte:

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

Eine gemischte Verknüpfung von 16- und 8-Bit-Werten ist nicht zulässig. Beim Zugriff auf Speichervariable 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.

### 3.3.5 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 Label1
```

prüft der Prozessor ob das Bit 0 in AH gesetzt ist. In diesem Fall wird das Zero-Flag gelöscht. Dadurch wird der folgende Sprungbefehl (JNZ Label 1) in Abhängigkeit vom Wert des Bits ausgeführt oder ignoriert. Tabelle 3.15 enthält eine Aufstellung gültiger TEST-Befehle.

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

*Tabelle 3.15: Die TEST-Befehle*

Bei Zugriffen auf reine Speichervariable benötigt der A86-Assembler die Schlüsselworte:

```
TEST WORD [BX+3000], 3000
TEST BYTE [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.

## 3.4 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 Shift-Operationen unterschieden. Arithmetische Shift-Operationen dienen zur Multiplikation (Shift links) und Division (Shift rechts) des Operanden um den Faktor  $2 * n$ , wobei n die Zahl der Shift-Operationen 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.

### 3.4.1 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 3.24).

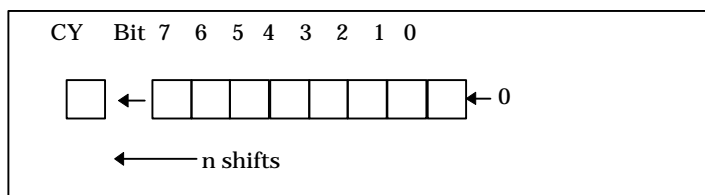


Bild 3.24 : Shift Left bei 8-Bit-Operanden

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 Shift-Operation zu Null gesetzt. Tabelle 3.16 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 BYTE [DI+1],1
SAL/SHL WORD [DI+1],CL

Tabelle 3.16: Die SAL/SHL-Befehle

Bei Zugriffen auf Speichervariable benötigt der Assembler die Schlüsselworte:

```
SHL BYTE [3000],1
SAL WORD [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.

### 3.4.2 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 3.25).

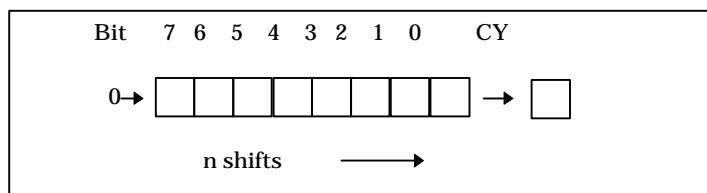


Bild 3.25: Shift Right bei 8-Bit-Operanden

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 Shift-Operation auf Null gesetzt. Tabelle 3.17 enthält eine Aufstellung gültiger SHR-Befehle.

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

Tabelle 3.17: Der SHR-Befehl

Bei Zugriffen auf Speichervariable benötigt der A86-Assembler die Schlüsselworte:

```
SHR BYTE [3000],1
SHR WORD [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.

### 3.4.3 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 3.26).

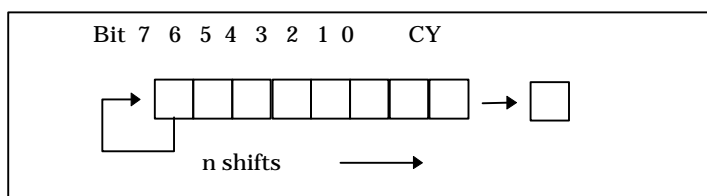


Bild 3.26 : Shift Arithmetic Right (SAR) bei 8-Bit-Operanden

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 Shift-Operation das oberste Bit erhalten und der Wert wird in das nächste rechts stehende Bit kopiert. Dadurch bleibt das Vorzeichen des Operanden erhalten. Tabelle 3.18 enthält eine Aufstellung gültiger SAR-Befehle.

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

Tabelle 3.18: Der SAR-Befehl

Der SAR-Befehl produziert nicht das gleiche Ergebnis wie ein IDIV-Befehl. Bei Zugriffen auf Speichervariable benötigt der A86-Assembler die Schlüsselworte:

```
SAR BYTE [3000],1
SAR WORD[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.

## 3.5 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.

### 3.5.1 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 3.27).

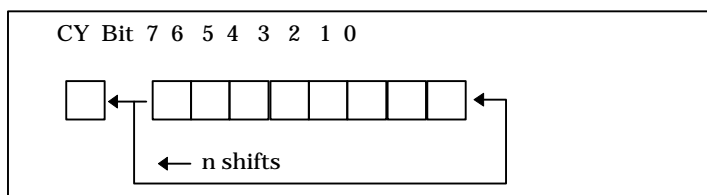


Bild 3.27: Der ROL-Befehl bei 8-Bit-Operanden

Der Befehl besitzt das Format:

ROL Ziel, Count

Count gibt dabei an, um wieviele Bitpositionen der Zielooperand 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 3.19 enthält eine Aufstellung gültiger ROL-Befehle.

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

Tabelle 3.19: Die ROL-Befehle

Als Operanden sind Register und Speichervariable erlaubt. Beim Zugriff auf Speicheradressen erwartet der A86 die Schlüsselworte BYTE oder WORD:

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

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

### 3.5.2 Der ROR-Befehl

Die Anweisung ROR (Rotate Right) arbeitet analog dem ROL-Befehl. Einziger Unterschied: Die Richtung der Rotation ist nach rechts gekehrt (Bild 3.28).

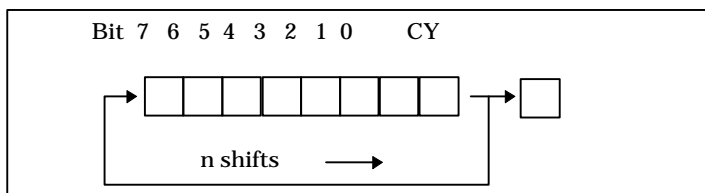


Bild 3.28: Die ROR-Operation bei 8-Bit-Operanden

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.

### 3.5.3 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 3.29).

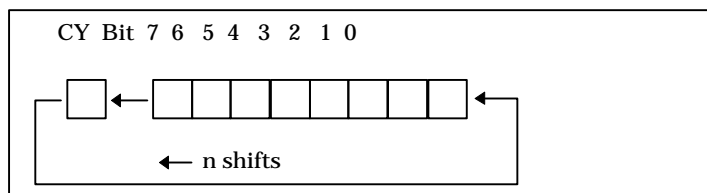


Bild 3.29: Der RCL-Befehl bei 8-Bit-Operanden

Der Befehl besitzt folgendes Format:

RCL Ziel, Count

Mit Ziel wird dabei der Zieloperand (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 WORD [DI+1],1
RCL BYTE [DI+1],CL

Tabelle 3.20: 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 3.20 enthält eine Aufstellung gültiger RCL-Befehle.

Beim Zugriff auf Speicheradressen erwartet der A86-Assembler die Schlüsselworte:

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

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

### 3.5.4 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 3.30).

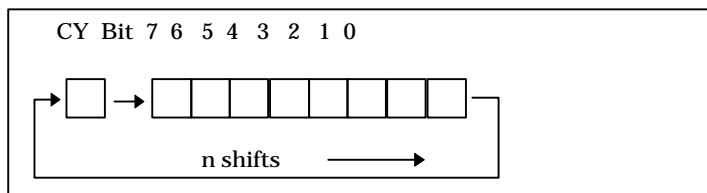


Bild 3.30: Der RCR-Befehl bei 8-Bit-Operanden

Es gilt die gleiche Syntax wie beim RCL-Befehl.

## 3.6 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.

### 3.6.1 Clear Carry-Flag (CLC)

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

```
CLC
```

und setzt das Carry-Flag zurück.

### 3.6.2 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.

### 3.6.3 Set Carry-Flag (STC)

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

STC

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

### 3.6.4 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.

### 3.6.5 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.

### 3.6.6 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.

### 3.6.7 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.

## 3.7 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.

### 3.7.1 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,3FFFH). 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 3.31).

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

Bild 3.31: Darstellung einer negativen Zahl

Bei einem gesetztem Bit (z.B. FFFFH) 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 3.32 in einem Wort speichern.

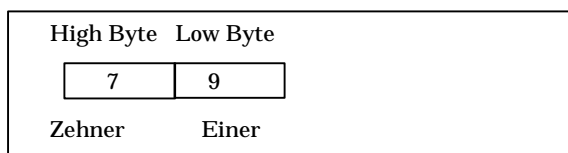


Bild 3.32: Speicherung einer ungepackten Dezimalzahl

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 3.32 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 3.33).

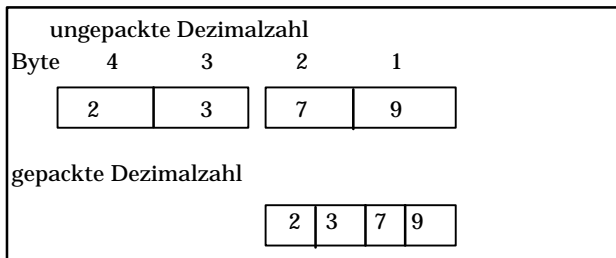


Bild 3.33: Darstellung von gepackten und ungepackten Dezimalzahlen

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 3.34 kodiert.

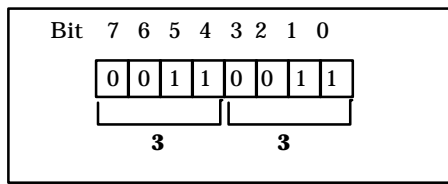


Bild 3.34: Darstellung einer BCD-Zahl

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 3.21).

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 3.21: 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.

### 3.7.2 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 Zieloperanden gespeichert. Der Befehl verändert die Flags:

AF, CF, OF, PF, SF, ZF

Als Operanden dürfen Speichervariable, Register und Konstante (nur Quelle) benutzt werden. Tabelle 3.22 enthält eine Übersicht gültiger 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 WORD [BX+10],3FF	Memory/Konst.
ADD BYTE [BX+10],3F	Memory/Konst.

Tabelle 3.22: Die ADD-Befehle

Bei Zugriffen auf Memoryadressen erwartet der A86 die Schlüsselworte BYTE oder WORD:

```
ADD WORD BP+02],0020
ADD BYTE [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.

### 3.7.3 Der ADC-Befehl

Der Befehl ADC (Add with Carry) addiert analog zu ADD den Wert zweier Operanden und legt das Ergebnis im Zielooperanden 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 3.23 gibt eine Übersicht über gültige 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 WORD [BX+10],3FF	Memory/Konst.
ADC BYTE [BX+10],3F	Memory/Konst.

*Tabelle 3.23: Die ADC-Befehle*

Bei Zugriffen auf Memoryadressen erwartet der A86-Assembler die Schlüsselworte BYTE oder WORD:

```
ADC WORD [BP+02],0020
ADC BYTE [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

Nachfolgend wird ein Programmfragment gezeigt, welches die beiden Befehle ADD und ADC verwendet, 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 3.6: 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.

### 3.7.4 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 Nibbles 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 3.7: BCD-Addition

Dieses Beispiel läßt sich in anderen A86-Programmen verwenden.

### 3.7.5 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 (obere 4 Bit) 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.

### 3.7.6 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 3.24 enthält eine Übersicht gültiger SUB-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 WORD [BX+10],3FF	Memory/Konst.
SUB BYTE [BX+10],3F	Memory/Konst.

Tabelle 3.24: Die SUB-Befehle

Bei Zugriffen auf Memoryadressen erwartet der A86-Assembler die Schlüsselworte BYTE oder WORD:

```
SUB WORD [BP+02],0020
SUB BYTE [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.

### 3.7.7 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

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 WORD [BX+10],3FF	Memory/Konst.
SBB BYTE [BX+10],3F	Memory/Konst.

Tabelle 3.25: Die SBB-Befehle

Tabelle 3.25 enthält eine Übersicht über gültige SBB-Befehle. Bei Zugriffen auf Memoryadressen erwartet der Assembler die Schlüsselworte:

```
SBB WORD [BP+02],0020
SBB BYTE [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 Wort- 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.

### Programmbeispiel

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 3.8: Subtraktion*

Das Ergebnis der Subtraktion findet sich in DX:AX.

### 3.7.8 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.

### 3.7.9 Der AAS-Befehl

Ähnlich dem AAA-Befehl wird die AAS-Anweisung (ASCII-Adjust for Subtraktion) 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.

### 3.7.10 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 3.26 gibt eine Auswahl gültiger MUL-Befehle an.

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

Tabelle 3.26: Die MUL-Befehle

Eine Multiplikation mit Konstanten ist nicht vorgesehen (die 8086-kompatiblen NEC V20 und V30 CPUs besitzen aber solche Befehle). Bei Zugriffen auf Memoryadressen muß angegeben werden, ob es sich um ein Byte oder ein Wort handelt:

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

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.

### Programmbeispiel

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 3.9: 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.

### 3.7.11 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 vorzeichenbehaftete 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 Speichervariable verwendet werden. Tabelle 3.27 gibt eine Auswahl gültiger IMUL-Befehle an.

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

*Tabelle 3.27: Die IMUL-Anweisung*

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

IMUL WORD [BP+02]  
IMUL BYTE [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.

### 3.7.12 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.

### 3.7.13 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 3.28 gibt einige gültige DIV-Befehle an.

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

Tabelle 3.28: Die DIV-Befehle

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

DIV WORD [BP+02]  
 DIV BYTE [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.

#### Programmbeispiel

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 3.10: Division*

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

### 3.7.14 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 3.29 gibt einige gültige Befehle an.

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

*Tabelle 3.29: Die IDIV-Befehle*

Bei Zugriffen auf Memoryadressen erwartet der Assembler die Schlüsselworte:

```

IDIV WORD [BP+02]
IDIV BYTE [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.

### 3.7.15 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.

### 3.7.16 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 3.30).

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

*Tabelle 3.30: 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]

Der Assembler erwartet beim indirekten Zugriff auf den Speicher die Schlüsselworte:

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

Die Programme sind gegebenenfalls an diese Nomenklatur anzupassen.

### 3.7.17 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 3.31 gibt einige gültige Befehlsformen an.

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

*Tabelle 3.31: Formen des INC-Befehls*

Der A86 erwartet jedoch beim Zugriff auf Speicherzellen die Schlüsselworte:

```
INC WORD [3000]
INC BYTE [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.

### Programmbeispiel

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
```

*Listing 3.11: Anwendung des INC-Befehls*

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.

### **3.7.18 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 3.32 gibt einen Überblick über mögliche Variationen.

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

*Tabelle 3.32: 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. Der A86 erwartet bei Speicherzugriffen folgende Schlüsselworte:

DEC WORD [BX+3]  
DEC BYTE [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 3.12: Anwendung des DEC-Befehls*

Das Beispiel läßt sich nur nachvollziehen, wenn es in ein A86-Programm eingefügt wird. Weitere Informationen finden sich im Abschnitt über die Sprungbefehle.

### 3.7.19 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

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

*Tabelle 3.33: Formen des NEG-Befehls*

und subtrahiert den Operanden von der Zahl 0. Dies bedeutet, daß der Operand durch Bildung des Zweierkomplements negiert wird.

Tabelle 3.33 gibt einige der möglichen Befehlsformen der NEG-Anweisung an. Der A86 benötigt zum Zugriff auf Speicherzellen die Schlüsselworte:

```
NEG WORD [BP+3]
NEG BYTE [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.

### 3.7.20 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.

### 3.7.21 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.

## 3.8 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 es 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.

### 3.8.1 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 3.35).

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

Bild 3.35: Die Wirkung des JMP-Befehls

In Bild 3.35 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.

**Anmerkung:** 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.

### Der JMP NEAR-Befehl

In Bild 3.35 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 Ziel

Das Prefix NEAR weist den Assembler an, einen 3-Byte-Befehl für einen Sprung innerhalb des aktuellen Codesegments zu generieren. 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 NEAR [1200]). Tabelle 3.34 enthält eine Aufstellung gültiger 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]

Table 3.34: JMP NEAR-Befehle

Bei absoluten Sprüngen kodiert der Assembler das Ziel als relativen Offset zur aktuellen Adresse. Bei der indirekten Adressierung wird das Sprungziel als absolute Offset-Adresse verarbeitet. Die Anweisungen:

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

veranlassen deshalb eine Programmverzweigung zur absoluten Adresse CS:1200. Solche Programme sind allerdings nicht mehr frei im Speicher verschiebbar. 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 `DS` genutzt, sofern `BP` nicht Verwendung findet. Dann erfolgt der Zugriff über das Stacksegment. Beispiel für die Verwendung der Sprungbefehle finden sich in den folgenden Listings.

**Anmerkung:** In Kapitel 2 finden Sie weitere Hintergrundinformationen zur Kodierung der Sprungbefehle.

## Der JMP SHORT-Befehl

Beim `JMP NEAR`-Befehl wird die Zieladresse mindestens mit 3 Byte codiert. Bei Sprüngen über weniger als 127 Byte kennt der 80x86 Prozessor eine weitere Variante, den `JMP SHORT`-Befehl, der mit 2 Byte codiert wird. Falls zu einer `JMP SHORT`-Anweisung das zugehörige Label außerhalb der Distanz von 127 Byte liegt, erzeugt der Assembler eine Fehlermeldung. Der `JMP SHORT`-Befehl besitzt die 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 FAR 1000:3FFF
JMP FAR WEITER
```

Mit FAR wird dem Assembler signalisiert, daß der Sprung über die Segmentgrenzen 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 oder indirekt als Speicheradresse:

```
JMP FAR [33FF]
JMP FAR [BX+DI+3]
```

angeben. Der Prozessor liest 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 auf Low Adresse) gespeichert ist. Eine Adressierung über Register ist nicht möglich. Das folgende Beispiel (Listing 3.13) zeigt die Verwendung verschiedener JMP-Befehle.

```
=====
; File : RESET.ASM (c) Born G.
; DOS System Restart auf BIOS-Adresse
; FFFF:0000. Programm als COM assemblieren
=====
;
;           RADIX 16           ; Hexadezimalsystem
;           ORG 0100           ; Startadresse COM
;           CODE SEGMENT
;
RESET: JMP NEAR Start          ; an Programmanfang
;
; Datenbereich mit dem Meldungstext
;
Text: DB "System Reset", 0A, 0D, "$"
;
Start: MOV AH,09               ; DOS-Textausgabe
;           MOV DX,OFFSET Text ; Textanfang
;           INT 21             ; DOS-Ausgabe
```

```

        JMP 0FFFF:0          ; BIOS-Restart
END Reset                          ; Programmende

```

Listing 3.13: RESET.ASM

Sprungbefehl	Beispiel
JMP SHORT Disp8	JMP SHORT Next
JMP NEAR Disp16	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 Label
JMP FAR Mem16	JMP FAR [1200]

Tabelle 3.35: Variationen des JMP-Befehls

Die Aufgabe des Programmes ist es, unter DOS einen Warmstart durchzuführen. Die Adresse der Warmstartroutine findet sich im BIOS-ROM ab Adresse FFFF:0000. Das Programm benutzt einen JMP FAR-Befehl zur entsprechenden Warmstartroutine. Tabelle 3.35 enthält nochmals eine Zusammenstellung aller Möglichkeiten des JMP-Befehls mit den verschiedenen Variationen.

## 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 abzutestenden Bedingung auszuführen sind. Dort ist 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 3.37 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 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 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 XOR OF) OR ZF) = 0	$X > Y$
JGE/ JNL	Jump if Greater or Equal Jump if Not Less	(SF XOR OF)=0	$X \geq Y$
JL/ JNGE	Jump if Less Jump if Not Greater nor Equal	(SF XOR OF)=1	$X < Y$
JLE/ JNG	Jump if Less or Equal Jump if Not Greater	((SF XOR OF) 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 3.36: 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. 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.

### **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.

### **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.

### **Programmbeispiel**

Das nachfolgende Beispiel aus Listing 3.14 nutzt die bedingten Sprungbefehle zur Erweiterung des DOS-Befehlssatzes. Das kleine Programm ASK.ASM erlaubt die Abfrage von Benutzereingaben aus DOS-Batchprogrammen. Beim Aufruf:

ASK Hallo

soll der Text *Hallo* auf dem Bildschirm erscheinen. Anschließend wird die Tastatur abgefragt und der Code des eingegebenen Zeichens an DOS zurückgegeben. Dieser Code läßt sich dann in Batchprogrammen durch den Befehl ERRORLEVEL abfragen (siehe Literaturhinweise /2/).

Bezüglich der Programmlogik möchte ich noch einige Informationen geben. 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. Die genaue Belegung des weitgehend undokumentierten PSP-Bereichs ist in /1/ (siehe Literaturhinweise) beschrieben. Der PSP-Bereich ist auch die Ursache dafür, daß ein Programm erst ab CS:100 (ORG 100) beginnen darf. Die Parameter der Kommandozeile finden sich gemäß Bild 3.14 folgende Struktur im PSP:

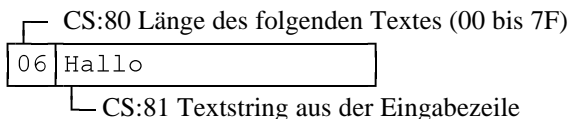


Bild 3.38: Aufbau des Transferpuffers im PSP-Bereich

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.

Die Ausgabe des Textes *Hallo* erfolgt mit der DOS-INT 21 Funktion AH = 02, die das Zeichen in DL erwartet. Die Tastatur läßt sich mit der INT 21-Funktion AH = 08H oder AH = 01 (mit Echo) abfragen, wobei der Zeichencode in AL zurückgegeben wird. Ist der Wert 00H, liegt ein Extended ASCII-Code einer Funktionstaste vor, dann ist ein zweites Zeichen von der Tastatur zu lesen. Der Code wird über die INT 21 Funktion AH = 4CH (DOS-Exit) im Register AL (Errorcode) übergeben. Um das Programm transparenter zu halten, wurde bereits ein Vorgriff auf den nachfolgend besprochenen CALL-Befehl getätigt.

```

;=====
; File: ASK.ASM    (c) Born G.
; Funktion: Programm zur Benutzerab-
; frage in Batchdateien. Aufruf:
;
;     ASK <Text>
;

```

```

; Der Text wird auf dem Screen ausge-
; gegeben. Der Tastencode wird an DOS
; zurückgegeben. (Bei Funktionstasten
; wird FF zurückgegeben). Er läßt sich
; per ERRORLEVEL abfragen. Das Programm
; ist als COM-Datei zu übersetzen!!
;=====
;
        RADIX 16          ; Hexadezimalsystem
        ORG 0100         ; Startadresse COM
        CODE SEGMENT
        Blank EQU 20     ; Blank
        Err1 EQU 0FF     ; Error
;
ASK:    JMP NEAR Start   ; ins Hauptprogramm
;
;-----
; Unterprogramm zur Textausgabe
;-----
; prüfe, ob Text im PSP vorhanden ist
;
Text:   CS: MOV CL,[80]   ; lies Pufferlänge
        CMP CL,0         ; Text vorhanden?
        JZ Ready        ; Nein -> Exit
;
; Text ist vorhanden, ausgeben per INT 21, AH = 02
;
        MOV BX,0081      ; Zeiger auf 1. Zeichen
Loop1:  ; Beginn der Ausgabeschleife !!!!
        MOV AH,02        ; INT 21-Code Display Char.
        CS: MOV DL,[BX]  ; Zeichen in DL laden
        INT 21          ; CALL DOS-Zeichenausgabe
        INC BX          ; Zeiger auf nächstes Zeichen
        DEC CL          ; Zeichenzahl - 1
;
        JNZ Loop1       ; Textende? Nein-> Loop
;
        MOV AH,02        ; INT 21-Code Display
        MOV DL,Blank     ; Blank anhängen
        INT 21          ; und ausgeben
Ready:  RET             ; Ende Unterprogramm
;
;-----
; Unterprogramm zur Tastaturabfrage
; benutze INT 21, AH = 08 Read Keyboard
; oder:          AH = 01 Read Keyboard & Echo
;-----
; lese 1. Zeichen
Key:    MOV AH,01        ; INT 21-Read Key & Echo
        INT 21          ; Read Code
        CMP AL,0        ; Extended ASCII-Code ?
        JNZ Exit        ; Nein -> Ready
;
; lies 2. Zeichen beim Extended ASCII-Code
;
        MOV AH,08        ; INT 21 Read Keyboard
        INT 21          ; Code aus Puffer lesen

```

```

        MOV AL,Err1          ; Fehlercode setzen
Exit:   RET                  ; Ende Unterprogramm

;
;-----
; Hauptprogramm
;-----
;
Start:  CALL NEAR Text       ; Textausgabe, falls vorhanden
        CALL NEAR Key        ; Abfrage der Tastatur
        PUSH AX              ; merke Tastencode
;
; CR,LF ausgeben
;
        MOV AH,09           ; INT 21-Stringausgabe
        MOV DX,Crlf         ; Stringadresse
        INT 21              ; ausgeben
        POP AX              ; restauriere Tastencode
;
; DOS-Exit, Returncode steht bereits in AL
;
        MOV AH,4C           ; INT 21-Exitcode
        INT 21              ; terminiere
;
; Bereich mit den Textkonstanten
;
Crlf:  DB 0D,0A,"$"
;
END Ask

```

Listing 3.14: Das Programm ASK.ASM

Das Programm muß als COM-Datei übersetzt werden und erlaubt die Erzeugung von komfortablen Menüoberflächen in DOS per Batchprogramm. Ein Beispiel für ein solches Batchprogramm findet sich im Anhang. Das Programm ASM.BAT stellt eine Entwicklungsumgebung für die Entwicklung mit A86 zur Verfügung. Es läßt sich leicht um weitere Punkte (z.B. Generiere OBJ-Files, Link der Files, etc.) erweitern.

### 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 Fehleroutine 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 obiger Tabelle aufgeführt. Die einzelnen Flags werden durch verschiedene Operationen (ADD, AND, CMP, etc.) gesetzt. Mit AND AX,AX läßt sich zum Beispiel prüfen, ob der Inhalt des Registers AX den Wert Null besitzt. Die Sprungbefehle verändern die Flags nicht. Bedingte Sprünge über eine Distanz größer 127 Byte lassen sich mit einem kleinen Trick erreichen:

```
.
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.

### 3.8.2 Die CALL-Befehle

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. Im Gegensatz zum JMP-Befehl merkt sich der Prozessor die Adresse, an der der CALL-Befehl gelesen wurde und setzt das Programm nach Beendigung des Unterprogrammes fort. 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

Das Schlüsselwort NEAR signalisiert dabei dem Assembler, daß es sich um einen Aufruf innerhalb des Segments handelt. Der CALL-NEAR-Befehl sichert nur den Inhalt des Instruktionpointers auf dem Stack und aktiviert dann das entsprechende Unterprogramm. Der CALL-NEAR-Befehl erlaubt verschiedene Adressierungsarten:

CALL NEAR 3000	; direkter CALL
CALL NEAR Key	; direkter CALL
CALL NEAR [3000]	; indirekt über den Speicher
CALL NEAR [BP+SI+2]	; indirekt über den Speicher
CALL NEAR AX	; Indirekt über Register

Am einfachsten ist die direkte Adressierung, bei der die Zieladresse als Konstante oder Label angegeben wird. 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 (z.B. CALL NEAR [BX+3000]). Die Zieladresse wird dabei standardmäßig aus dem Datensegment - eine Ausnahme bildet BP mit dem Zugriff auf SS - gelesen. Durch einen Segment-Override-Befehl läßt sich diese Einstellung allerdings überschreiben. Die Sequenz:

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

führt damit einen Sprung zur Adresse CS:2000 aus. Manche Assembler verlangen bei der indirekten Adressierung die Schlüsselworte WORD oder BYTE (z.B. CALL

WORD [BX+10]). Beispiele für die Anwendung des CALL-Befehls finden sich im Programm ASK.ASM.

### Der CALL FAR-Befehl

Soll ein Unterprogrammaufruf über die Segmentgrenzen hinaus erfolgen, ist der CALL FAR-Befehl mit folgender Syntax zu nutzen:

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 ist dagegen nicht möglich. Nachfolgend sind einige Befehle aufgeführt.

```
CALL FAR 3FFF:0100 ; direkt über absolute Adressen
CALL FAR Bios      ; direkt über Labels
CALL FAR [3000]    ; indirekt über Speicherzellen
CALL FAR [BX+SI+20] ; indirekt über Speicherzellen
```

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 (4-Byte-Rückkehradresse) dem Stack. Bei der indirekten Adressierung ist das Schlüsselwort DWORD erforderlich (z.B. CALL FAR DWORD [3000]).

### Der RET-Befehl

Im Programm ASK.ASM wurde bereits ein neuer Befehl zum Beenden der Unterprogramme eingeführt. Sobald im Programm ein RET auftaucht, liest die CPU die Rückkehradresse vom Stack und setzt das unterbrochene Hauptprogramm fort. Da es zwei verschiedene CALL-Befehle (NEAR, FAR) gibt, existieren auch die entsprechenden Rückkehrbefehle:

```
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 wechselweise Aufruf eines Unterprogrammes über CALL FAR und CALL NEAR ist damit nicht möglich.

Im Zusammenhang mit dem RET-Befehl möchte ich noch auf eine weitere Eigenschaft hinweisen. Oft werden dem Unterprogramm Parameter vom rufenden Programm übergeben, die bei der Rückkehr vom Stack zu entfernen sind - mit POP ist dies eine aufwendige Sache. Der RET-Befehl erlaubt deshalb optional einen Parameter (z.B. RET 2), der die Zahl der zu entfernenden Worte spezifiziert. Zuerst wird die Rückkehradresse gelesen und dann der Stackpointer um n Worte erhöht.

### 3.8.3 Der INT-Befehl

Dieser Befehl ermöglicht den Aufruf einer besonderen Art von Unterprogrammen - den Interrupt-Service-Routinen. Beispiele kennen Sie bereits durch den Aufruf der DOS-Routinen per INT 21. Jeder Software-Interrupt (INT xx) löst eine Programmunterbrechung aus. Die CPU sichert dann den Inhalt des Flagregisters und die Rücksprungadresse (CS:IP) auf dem Stack (also 6 Byte). Dann liest der Prozessor einen 4-Byte-Vektor mit der Zieladresse der Interrupt-Service-Routine aus einer Tabelle im Bereich zwischen 0000:0000 und 0000:03FFF ein und verzweigt zu dieser Adresse. Die genaue Adresse des Vektors läßt sich zu Interrupt-Nummer \* 4 errechnen. Die 8086-CPU kann 256 verschiedene Interrupts unterscheiden. Für jeden Interrupt existiert ein 4-Byte-Vektor in der Vektortabelle. Das DOS-Betriebssystem benutzt zum Beispiel den INT 21-Vektor, um Anwenderprogrammen Systemroutinen zur Verfügung zu stellen. Die Vektoren müssen entweder durch DOS oder die Anwenderprogramme initialisiert werden. Anschließend genügt zum Aufruf der Service-Routine ein softwaremäßiger INT xx-Befehl, der entgegen dem CALL-Befehl mit nur 2 Byte kodiert wird. Bei 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 Hardwareinterrupt verarbeitet. Der Aufruf:

```
MOV AH,4C    ; DOS Exit Code
MOV AL,00    ; ERRORLEVEL Code
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 in /1/ (siehe Literaturhinweise). Zwei Interrupts (INTO und INT 3) nehmen eine Sonderstellung ein. Der INTO-Befehl wird nur ausgeführt, wenn das Overflow-Flag gesetzt ist. Dies kann bei der Anwendung arithmetischer Befehle nützlich sein. Der INT3 wird durch ein Opcodebyte kodiert, was von Debuggern für Unterbrechungspunkte genutzt wird.

## Der IRET-Befehl

Eine Interruptroutine darf nicht mit einem einfachen RETF-Befehl beendet werden, da sonst der Inhalt des Flagregisters auf dem Stack verbleibt. Die Anweisung IRET sorgt nicht nur für die korrekte Restaurierung des Stacks, sondern restauriert auch den Inhalt des Flagregisters.

```

;=====
; File: INT.ASM (c) Born G.
; Funktion: Demonstration des INT Befehls für
;           BIOS- und DOS-Zugriffe. Das Programm
;           ist als COM-Datei zu übersetzen!!
;=====
;
;           RADIX 16          ; Hexadezimalsystem
;           ORG 0100         ; Startadresse COM
;           CODE SEGMENT
;           Blank EQU 20     ; Blank
;
;=====
; 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
;=====
;
Start:  MOV  AX,0600          ; up scroll, clear window
        MOV  BH,07          ; Attibut 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,0F0         ; Attibut 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,0916        ; Spalte/Zeile

```

```

                INT 10          ; BIOS Routine rufen
;
;=====
;  Schreibe String auf dem Schirm
;=====
;
                CALL NEAR String ; Ausgabe
;
; Rückkehr zu MS-DOS
;
                MOV  AX,4C00     ; DOS-Code "Exit"
                INT  21         ; Terminiere Programm
;
;=====
;  Unterprogramm zur Ausgabe eines Textes
;=====
;
String:  MOV  DX,OFFSET Txt ; lade Stringadresse
         MOV  AH,09        ; DOS-Code "Write String"
         INT  21         ; String ausgeben
         RET              ; Ende Unterprogramm
;
;
;=====
;  Datenbereich mit dem Textstring
;=====
Txt:  DB  "Assemblerprogrammierung mit dem A86",0D,0A,"$"
;
                END

```

Listing 3.15: Demonstration des INT-Befehls

## 3.9 Befehle zur Konstruktion von Schleifen

Neben den JCXZ-Befehl kennt der 8086-Prozessor weitere Anweisungen zur Konstruktion von Schleifen. Die LOOP-Befehle benutzen dabei das Register CX als Zähler und können SHORT-Sprünge über die Distanz von + 127 und -128 Byte ausführen.

### 3.9.1 Der LOOP-Befehl

Der Befehl besitzt die Syntax:

LOOP SHORT Label

und decremientiert (erniedrigt) bei jeder Ausführung den Inhalt des Registers CX um 1. 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 LOOP-Befehl beeinflusst die Flags nicht.

### 3.9.2 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:  $(CX \lessgtr 0$  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. Die beiden Bezeichnungen LOOPE (Loop While Equal) und LOOPZ (Loop While Zero) erzeugen den gleichen Befehlscode. Der LOOPE/LOOPZ-Befehl verändert selbst keine Flags.

### 3.9.3 Der LOOPNE/LOOPNZ-Befehl

Der LOOPNE/LOOPNZ-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:  $(CX \lessgtr 0$  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, wie oft die Schleife maximal durchlaufen werden darf. Ist das Zero-Flag vorher auf 1 gesetzt, wird die Schleife sofort beendet. Die beiden Bezeichnungen LOOPNE (Loop While Not Equal) und LOOPNZ (Loop While Not Zero) erzeugen den gleichen Befehlscode. Der LOOPNE/LOOPNZ-Befehl verändert selbst keine Flags.

## 3.10 Die String-Befehle

Die 80X86-Prozessorfamilie besitzt einen Satz von 5 Befehlen zur Bearbeitung von Strings (Byte- oder Wortfolgen) 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-Flags (s. Befehlsbeschreibung) bestimmt.

### 3.10.1 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 zu plazieren. Die CPU wertet dann den Inhalt des CX-Registers aus und wiederholt den nachfolgenden String-Befehl solange bis der Wert des Registers 0 annimmt.

### 3.10.2 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 Stringelement. Durch Kombination mit der REP-Anweisung läßt sich ein ganzer Speicherbereich verschieben.

### 3.10.3 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.

### 3.10.4 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.

### 3.10.5 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.

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

Mit dieser Anweisung wird das durch ES:DI adressierte Byte oder Wort durch den 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 Stringelements. Der Befehl läßt sich zusammen mit den REPEAT-Anweisungen recht elegant zur Initialisierung von kompletten Datenbereichen benutzen.

### 3.11 Der HLT-Befehl

Dieser Befehl veranlaßt den Wechsel der CPU in den HALT-Mode. Damit werden keine neuen Befehle mehr ausgeführt. Der Mode läßt sich durch einen Reset oder einen Hardwareinterrupt beenden.

### 3.12 Der LOCK-Befehl

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

### 3.13 Der WAIT-Befehl

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

### 3.14 Der ESC-Befehl

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

### 3.15 Die NEC V20/V30-Befehle

Die V20/V30-Prozessoren der Firma NEC sind pin- und softwarekompatibel zu den Intel 8088/8086-Prozessoren. Allerdings besitzen die NEC-Prozessoren einen erweiterten Befehlssatz. So sind sie in der Lage, den kompletten Befehlssatz der Intel 8080-Prozessoren zu verarbeiten. Der A86 ist einer der wenigen Assembler, der den NEC-Befehlssatz der V20/V30-Prozessoren unterstützt. Nachfolgend werden die Erweiterungen zu 8086-Befehlssatz kurz angerissen. Die Registerbezeichnung der NEC-Prozessoren weicht von der INTEL-Bezeichnung ab (z.B. IX = SI, IY = DI). Nachfolgend wird aber die INTEL-Bezeichnung SI/DI benutzt.

### ADD4S

Dieser Befehl erlaubt explizit die Addition von gepackten BCD- Zahlen. Die beiden V20/V30-Indexregister SI und DI dienen dabei als Zeiger auf die zwei BCD-Werte. Die gepackte BCD-Zahl an der durch DS:SI adressierten Speicherstelle wird auf den durch ES:DI adressierten BCD-Wert addiert und gespeichert. Die Länge des BCD-Strings in Ziffern ist im Register CL zu übergeben. Der Befehl beeinflusst die Flags OF, CY und ZF.

### CALL80 imm8

Dieser Befehl ist nur im 8080-Modus möglich und aktiviert über den Interrupt Nummer *imm8* eine Service-Routine mit 8080-Befehlen. Die Routine ist durch einen RETI-Befehl zu verlassen.

### CLRBIT Op1,Op2

Mit diesem Befehl lassen sich die in Op2 angegebenen Bits im Operanden Op1 löschen. Der Befehl kennt verschiedene Variationen.

```
CLRBIT DL,CL
CLRBIT BYTE [BX+10],CL
CLRBIT DX,CL
CLRBIT WORD [BX+10],CL
CLRBIT DL,01
CLRBIT BYTE [BX+10],02
CLRBIT AX,01
CLRBIT WORD [BX+10],80
```

Als *Op1* dürfen sowohl 8-Bit- als auch 16-Bit-Register und Memoryvariable verwendet werden. *Op2* enthält das Bitmuster entweder als 8-Bit-Konstante, oder es wird der Inhalt des Registers CL benutzt. Der Befehl verändert keine Flags.

### CMP4S

Dieser Befehl arbeitet wie die SUB4S-Anweisung und erlaubt einen Vergleich (compare) von gepackten BCD-Zahlen. Die BCD-Werte werden aber durch die Operation nicht verändert. Die gepackte BCD-Zahl an der durch DS:SI adressierten Speicherstelle wird mit der Zahl in ES:DI durch Subtraktion verglichen. Die Länge des BCD-Strings in Ziffern ist im Register CL zu übergeben. Der Befehl beeinflusst die Flags OF, CY und ZF.

### LODBITS Op1, Op2

Dieser Befehl lädt das AX-Register mit einem Bitfeld. Der Beginn des Bitfeldes wird durch die Register DS:SI adressiert. Die unteren 4 Bit des Op1 (8-Bit-Register) geben den Bit-Offset in das Bitfeld an. Die Länge des Bitfeldes steht im Operanden 2 (8-Bit-

Register oder 8-Bit-Konstante). Nach der Befehlsausführung werden SI und die unteren 4-Bit des Operanden 1 automatisch erhöht, so daß sie auf das nächste Bitfeld zeigen.

### **NOTBIT Op1,Op2**

Der Befehl besitzt die gleiche Syntax wie die Anweisung CLRBIT, invertiert aber die spezifizierten Bits.

### **REPC/REPNC**

Dies ist ein LOOP-Befehl, der zusammen mit Stringbefehlen (CMPS, SCAS) verwendet werden kann. Er wird solange ausgeführt, bis entweder CX = 0 ist oder das Carry-Flag gelöscht wird (REPC) oder das Carry-Flag gesetzt wird (REPNC).

### **ROL4 Op1**

Diese Anweisung rotiert 4 Bit (1 Nibble) des angegebenen Operanden (8-Bit-Register oder 8-Bit-Memoryvariable) mit dem Inhalt von AL. Dabei werden alle Bits von Op1 um 4 Positionen nach links geschoben. Die herausfallenden Bits werden von links in AL in die unteren Positionen eingesetzt. Gleichzeitig werden die unteren 4 Bits von AL in das untere Nibble von Op1 transferiert.

### **ROR4 Op1**

Dieser Befehl arbeitet wie ROL4 mit dem einzigen Unterschied, daß die Bits um 4 Positionen nach rechts verschoben werden.

### **SETBIT Op1,Op2**

Der Befehl besitzt die gleiche Syntax wie die Anweisung CLRBIT, setzt aber die angegebenen Bits.

### **STOBITS Op1, Op2**

Dieser Befehl speichert die Bits im AX-Register in ein Bitfeld. Der Beginn des Bitfeldes wird durch die Register ES:DI adressiert. Die unteren 4 Bit des Op1 (8-Bit-Register) geben den Bit-Offset in das Bitfeld an. Die Länge des Bitfeldes steht im Operanden 2 (8-Bit-Register oder 8-Bit-Konstante). Nach der Befehlsausführung werden DI und die unteren 4-Bit des Operanden 1 automatisch erniedrigt, so daß sie auf das nächste Bitfeld zeigen.

### **SUB4S**

Dieser Befehl erlaubt explizit die Subtraktion von gepackten BCD-Zahlen. Der gepackte BCD-Wert an der durch DS:SI adressierten Speicherstelle wird von der an der durch ES:DI adressierten BCD-Zahl subtrahiert und nach ES:DI gespeichert. Die

Länge des BCD-Strings in Ziffern ist im Register CL zu übergeben. Der Befehl beeinflusst die Flags OF, CY und ZF.

### **TESTBIT Op1,Op2**

Der Befehl besitzt die gleiche Syntax wie die Anweisung CLRBIT, prüft aber die angegebenen Bits auf Übereinstimmung mit dem Muster. Der Befehl setzt das Zero-Flag, falls das getestete Bit den Wert 0 besitzt.

Zudem unterstützt der V20/V30-Befehlssatz bereits die 80186/ 80286 Befehle (PUSH imm16, PUSHA, POPA, MUL imm16, SHL imm8, SHR imm8, SAR imm8, SAL imm8, ROL imm8, ROR imm8, RCL imm8, RCR imm8, CHKIND, INS, OUTS, ENTER, LEAVE).

## **3.16 Die 80186/80286 Befehlerweiterungen**

Der 80186 und der 80286-Prozessor besitzen einige erweiterte Befehle, die bereits vom A86 unterstützt werden. Nachfolgend möchte ich diese Befehle kurz vorstellen.

### **ARPL Op1,Op2 (80286)**

Der Befehl besitzt zwei Operanden und justiert das RPL Feld mit den Selektoren. Der erste Operand ist eine 16-Bit-Memoryvariable oder ein 16-Bit-Register mit dem Wert des Selektors. Der zweite Operator ist ein 16-Bit-Register. Sind die RPL-Bits (untere 2 Bits) des Selektors in Op1 kleiner als der Wert in Op2, wird das Zero-Flag auf 1 gesetzt. Dann wird das RPL-Feld auf den Wert von OP2 erhöht. Der Befehl dient zur Begrenzung der Zugriffsprivilegien im Protected Mode des 80286.

### **BOUND Op1,Op2 (80186/80286)**

Dieser Befehl erlaubt die Prüfung, ob ein Feldindex innerhalb der gültigen Grenzen liegt. Op2 adressiert dabei einen Memorydescriptor (2 16-Bit-Werte). Der Wert in Op1 (16-Bit-Register) muß größer als der Wert des ersten Memorywortes und kleiner als der Wert des zweiten Memorywortes sein. Liegt der Wert außerhalb dieser Grenzen, wird ein INT 5 ausgelöst. Der Befehl steht im REAL-Mode (unter DOS) nicht zur Verfügung.

### **CLTS (80286)**

Der Befehl Clear Task Switched Flag steht nur im Protected Mode des 80286 zur Verfügung und löscht das betreffende Flag. Das Flag kann nur in der Privileg Ebene 0 gelöscht werden.

**ENTER Op1, Op2 (80186/80286)**

Der Befehl erzeugt auf dem Stack einen Block (Frame) von n Bytes für lokale Variable bei Hochsprachen. Die Zahl der Bytes wird in Op1 übergeben (16-Bit-Konstante). Der zweite Operand gibt die Schachtelungstiefe für die Stackframes an (8-Bit-Konstante). Ist der Wert gleich 0, wird der Inhalt von BP auf dem Stack gespeichert und das Register mit dem Wert von SP geladen. Das Register BP wird zur Adressierung der lokalen Frames genutzt.

**IMUL Op1,Op2 (80186/80286)**

Im 8086-Mode führt der Befehl die Operation  $DX:AX := AX * DX$  aus. Mit IMUL Op1, Op2 (z.B. IMUL BX,30) wird die 8- oder 16-Bit Konstante in Op2 mit dem Inhalt von Op1 multipliziert. Das Ergebnis findet sich immer im angegebenen 16-Bit-Register Op1. Überschreitet das Ergebnis den 16-Bit-Bereich, wird ein Overflow ausgelöst. Als Besonderheit sind 3-Operanden-Befehle zulässig. In Op1 ist das 16-Bit-Zielregister anzugeben. In Op2 wird die Memoryadresse (Byte oder Word) des ersten Multiplikanten spezifiziert. Die Anweisung IMUL BYTE BX,SI,5 multipliziert den Inhalt des Bytes ab Adresse DS:SI mit der Bytekonstanten 5 und speichert das Ergebnis in BX ab. IMUL WORD BX,SI,2000 multipliziert eine Wordkonstante mit einer Speicherzelle (DS:SI) und speichert das Ergebnis in BX.

**INSB/INSW (80186/80286)**

Die Befehle INSB (Input String Byte) und INSW (Input String Word) lesen ein Byte oder Word aus dem im Register DX angegebenen Port in den Speicher (String) mit der Adresse ES:DI. Nach dem Befehl wird DI um den Wert 1 (Byte) oder 2 (Word) erhöht oder erniedrigt - abhängig vom Direction Flag). Mit der REP-Anweisung lassen sich Strings aus einem Port einlesen.

**LAR (80286)**

Mit diesem Befehl (Load Access Right Byte) wird im 2. Operand eine 16-Bit-Speicherzelle oder ein 16-Bit-Register mit dem Selektor angegeben. Ist der Selektor mit den aktuellen Privilegien gültig, wird der Descriptor in das High Byte des ersten Operanden (Register) geladen. Eine ausgeführte Operation signalisiert die CPU mit einem gesetzten Zero-Flag.

**LEAVE (80186/80286)**

Dieser Befehl ist das Gegenstück zu ENTER und kopiert den Inhalt von BP in SP. Damit wird der reservierte Stack wieder freigegeben. Anschließend wird BP mit dem alten Wert vom Stack restauriert.

**LGDT/LIDT (80286)**

Der Befehl LGDT lädt das Global Descriptor Table Register mit den 6 Byte des angegebenen Effective Address Operanden aus dem Speicher. Der Befehl LIDT führt die gleiche Operation auf der Interrupt Descriptor Tabelle aus.

**LLDT (80286)**

Dieser Befehl lädt das *Local Descriptor Table Register* mit dem durch den Memoryoperanden angegebenen Speicherinhalt.

**LMSW (80286)**

Dieser Befehl lädt das Maschine Status Word mit dem Wert des angegebenen Operanden. Dadurch läßt sich zum Beispiel der Protected Mode aktivieren.

**LSL (80286)**

Der Befehl lädt das Segment Limit des im 2. Operanden (Register oder Memory) angegebenen Selektors in den ersten Register-Operanden.

**LTR (80286)**

Der Befehl Load Task Register lädt das betreffende Register mit dem Inhalt des Operanden (Register oder Speicherwort).

**OUTSB/OUTSW (80186/80286)**

Die Befehle OUTSB (Output String Byte) und OUTSW (Output String Word) schreiben ein Byte oder Word in den im Register DX angegebenen Port. Quelle ist der Speicher (String) mit der Adresse ES:DI. Nach dem Befehl wird DI um den Wert 1 (Byte) oder 2 (Word) erhöht oder erniedrigt - abhängig vom Direction Flag. Mit der REP-Anweisung lassen sich Strings aus einem Port einlesen.

**POPA (80186/80286)**

Der Befehl restauriert die Register DI, SI, BP, SP, BX, DX, CX und AX vom Stack.

**PUSH imm (80186/80286)**

Mit dieser Instruktion wird der PUSH-Befehl der 8088/8086-Prozessoren erweitert, es läßt sich eine Byte- oder Word-Konstante (z.B. PUSH 3FFFH) auf den Stack speichern. Bei Bytekonstanten wird diese vorher auf 16-Bit mit dem korrekten Vorzeichen erweitert.

**PUSHA (80186/80286)**

Der Befehl sichert die Register AX, CX, DX, BX, SP, BP, SI und DI auf dem Stack.

**RCL/RCR/ROL/ROR/SAL/SAR/SHL/SHR (80186/80286)**

Ab dem 80186 läßt sich die Zahl der zu rotierenden Bits auch als Konstante angeben, falls der Wert größer 1 ist (z.B: RCL AX,3). Beim 8086 ist nur der Wert 1 für die Zahl der Rotationsschritte erlaubt.

**SGDT/SIDT (80286)**

Der Befehle SGDT kopiert den Inhalt des Global Descriptor Table Registers in die Global Descriptor Table im Speicher (6 Byte). Der Befehl SIDT führt die gleiche Operation auf der Interrupt Descriptor Table aus.

**STR (80286)**

Der Befehl sichert das Task-Register in das Wort an der angegebenen Speicheradresse.

**VERR/VERW (80286)**

Die Befehle Verify Segment for Read (VERR) und Verify Segment for Write (VERW) erwarten ein 16-Bit-Register oder einen Speicheroperanden, welches den Wert des Selektors enthält. Falls das Segment gelesen oder beschrieben werden darf, wird das Zero Flag gesetzt.

Diese Befehle wurden nur kurz besprochen, da sie unter DOS nicht zur Verfügung stehen. Weitere Informationen sind der Literatur über die 80186/80286-Prozessoren zu entnehmen.

## 3.17 Die 80x87-Fließkommabefehle

Der A86 unterstützt die Fließkommabefehle der Arithmetikprozessoren 8087 und 80287 mit der +F-Option. Anschließend wird vor jedem Fließkommabefehl ein FWAIT-(WAIT)-Befehl generiert. Aus Platzgründen möchte ich an dieser Stelle auf die Beschreibung des Numerik-Befehlssatzes verzichten. Hinweise finden sich in der A86-Dokumentation und in den Datenbüchern der Prozessorhersteller.



## 4 Die A86-Directiven

Zur Steuerung des Assemblers lassen sich eine Reihe von Pseudobefehlen im Programm angeben. Einige dieser Befehle haben Sie bereits in den Beispielprogrammen aus Kapitel 3 kennengelernt (z.B. RADIX 16). Diese Befehle generieren keinen Code, sondern dienen zur Steuerung des Übersetzervorgangs. Nachfolgend möchte ich kurz auf diese Pseudobefehle (Directiven) eingehen. Weitere Hinweise finden sich in der Originaldokumentation des A86.

### 4.1 Die Darstellung von Konstanten

Im Assemblerprogramm lassen sich Konstante innerhalb von Anweisungen (z.B. MOV AX,3FFF) eingeben. Dabei dürfen die Konstanten in verschiedenen Zahlensystemen eingegeben werden. Der A86 benutzt einige implizite Konventionen: ist die erste Ziffer eine 0, handelt es sich um eine Hexadezimalzahl. Dezimalzahlen dürfen keine vorangestellte 0 aufweisen. Diese Einstellung läßt sich durch ein angehängtes Zeichen überschreiben. Mit B oder xB sind Binärzahlen zu markieren, Mit O oder Q können oktale Zahlen kodiert werden. Ein angehängtes H signiert eine gültige Hexadezimalzahl und D oder xD eine gültige Dezimalzahl. Um eine eindeutige Unterscheidung von Hexadezimalzahlen zu erreichen, sollte xB und xD als Postfix verwendet werden.

### 4.2 Die RADIX-Directive

Die obigen impliziten Definitionen für die Interpretation der Zahlenbasis läßt sich mit dem RADIX-Kommando überschreiben. Die Anweisung besteht aus dem Schlüsselwort und einer Zahl, die die gewünschte Zahlenbasis spezifiziert: RADIX xx. Die folgenden Anweisungen verdeutlichen die Anwendung des Befehls:

```
RADIX 10      ; alle Werte als Dezimalzahlen lesen
DB 12,13,1010 ; Wert = 12, 13, 1010 (dezimal)
RADIX 16      ; alle Werte als Hexadezimalzahlen lesen
DB 12,13,101  ; Werte 18, 19, 257 (dezimal)
RADIX 2       ; alle Werte als Binärzahlen lesen
DB 10,100,1010 ; Werte 2, 4 , 10 (dezimal)
```

Dem RADIX-Kommando darf aus Kompatibilitätsgründen zu anderen Assemblern ein Punkt (.RADIX 16) vorangestellt werden. Standardmäßig liest der A86 alle Zahlen als Dezimalzahlen. Mit dem Parameter +D läßt sich diese Einstellung beim Aufruf auch

explizit setzen. Zahlen, die allerdings eindeutig einer anderen Zahlenbasis (z.B. 013F) zuzuordnen sind, werden aber durch die RADIX-Anweisung nicht beeinflusst. In den Beispielprogrammen wird die RADIX 16-Anweisung verwendet, um alle Eingaben als Hexzahl zu lesen.

## 4.3 Der HIGH/LOW-Operator

Mit diesem Operator läßt sich ein Byte einer Wortkonstanten extrahieren. Mit der Anweisung:

```
MOV AH,HIGH(04C00)
```

wird der Wert 04CH vom A86 in den Befehlscode eingesetzt. Mit der Option LOW läßt sich das untere Byte eines Wortes (z.B. `CMP AL,LOW(0FF00)`) extrahieren. Die Operatoren sind dann interessant, wenn 16-Bit-Konstanten global (z.B. mit EQU) definiert wurden, aber nur das obere oder untere Byte in einem Ausdruck verwendet werden soll.

## 4.4 Der BY-Operator

Dieser Operator wird nur vom A86 unterstützt und sollte aus Kompatibilitätsgründen nicht verwendet werden. Er erlaubt es, zwei Bytewerte zu einem Wort zu kombinieren (Operand1 BY Operand2). Vorstellbar ist diese Konstellation beim Laden eines 16-Bit-Registers (z.B. `MOV AX, Exit BY Error`, wobei Exit EQU 04C und Error EQU 00 definiert ist).

## 4.5 Operationen auf Ausdrücken

Bei der Erstellung von Assemblerprogrammen werden häufig Konstanten in Ausdrücken verwendet (z.B.: `AND AX,3FFF`). Die Konstante ist dabei in der geeigneten Form im Programm anzugeben. Eine Möglichkeit besteht darin, als Programmierer den Wert der Konstanten zu berechnen und im Quellprogramm einzusetzen. Dies ist aber nicht immer erwünscht: so kann es durchaus fehlerträchtig sein, wenn mehrere Werte manuell addiert werden. Um das Programm möglichst transparent zu gestalten, sollte man die ursprünglichen Teilwerte im Programm mit angeben. Die meisten Assembler unterstützen diese Form und berechnen zur Übersetzungszeit den Wert eines Ausdruckes. Der A86 bietet eine Reihe solcher Operatoren, die nachfolgend kurz aufgeführt werden.

### 4.5.1 Addition

Der Operator erlaubt die Addition mehrerer Konstanten innerhalb eines Ausdrucks. Die Addition kann dabei durch ein Pluszeichen, einen Punkt, durch das Schlüsselwort PTR oder zwei Operanden erfolgen. Die Beispiele verdeutlichen diesen Sachverhalt:

```
CR EQU 0A
DB 100 + CR      ; Addiere Konstante auf den Wert 100
DD 100.0 + 27.0 ; Additon zweier Fließkommazahlen
MOV AX,04C00 + 22 ; Berechne Konstante
```

Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen sowie Kommazahlen verwendet werden.

### 4.5.2 Subtraktion

Der Operator erlaubt die Subtraktion von Konstanten innerhalb eines Ausdrucks. Die Subtraktion wird dabei durch das Minuszeichen markiert (z.B. MOV AX,033-030). Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen sowie Kommazahlen verwendet werden. Bei Variablen muß der Typ der beiden Operatoren übereinstimmen.

### 4.5.3 Multiplikation und Division

Die Operatoren erlauben die Multiplikation und Division von Konstanten innerhalb eines Ausdrucks. Die Operatoren dürfen nur mit Kommazahlen oder ganzen Zahlen durchgeführt werden.

```
CMP CL, 2 * 3      ; Compare mit 6
MOV DX, 256 / 16   ; lade DX mit 16
DW 322 MOD 2
```

### 4.5.4 Schiebeoperatoren

Mit den Operatoren SHR, SHL und BIT lassen sich Schiebeoperationen auf einer Konstanten oder einem Ausdruck ausführen. Die Operatoren besitzen das Format:

```
Operand SHR Count (shift right)
Operand SHL Count (shift left)
BIT Count (bit Nummer)
```

Die Schiebepfehle erlauben es, den Operanden bitweise nach links oder rechts zu verschieben. Der zweite Operand *Count* gibt dabei die Zahl der zu verschiebenden

Binärstellen an. Die Bits, die in den Operanden eingeschoben werden, sind mit dem Wert 0 belegt. Beispiele für die Anwendung der Operatoren sind:

```
MOV BX,0FF33 SHR 4 ; BX = 0FF3
MOV DX,01 SHL 4 ; DX = 010
AND AL,BIT 1 ; AND AL,01
```

Mit dem Operator BIT läßt sich ein einzelnes Bit setzen.

## 4.5.5 Die logischen Operatoren

Mit den Anweisungen AND, OR, XOR und NOT lassen sich logische Operationen auf den Operanden ausführen. Die Operatoren besitzen das Format:

```
Operand AND Operand
Operand OR Operand
Operand XOR Operand
NOT Operand
```

Die Befehle dürfen ausschließlich mit vorzeichenlosen Byte- oder Word-Konstanten benutzt werden. Die Anweisung:

```
MOV AL,03F AND 0F
```

blendet zum Beispiel die oberen 4 Bits der Konstanten 3FH aus. Für die Verknüpfung der Operanden gelten die Regeln für AND, OR, XOR und NOT. Der NOT-Operator invertiert den Wert der angegebenen Konstanten.

## 4.5.6 Der NEG-Operator

Mit dem Operator *! Operand* läßt sich ein Wert negieren. Ist der Wert des Operators größer 0 (irgendein Bit gesetzt), wird als Ergebnis 0 zurückgegeben. Ist der Operand gleich 0, wird der Wert 0FFFF zurückgegeben. Mit dem Operator läßt sich also ein boolescher Vergleich durchführen.

## 4.5.7 Die Vergleichsoperatoren

Der A86 bietet einen weiteren Set an Vergleichsoperatoren:

```
Operand EQ Operand (equal)
Operand NE Operand (not equal)
Operand LT Operand (less than)
Operand LE Operand (less or equal)
Operand GT Operand (greater than)
Operand GE Operand (greater than or equal)
```

Als Operanden müssen vorzeichenlose Ganzzahlen (Byte oder Word) angegeben werden. Dabei können sowohl Konstanten als auch Variablen benutzt werden (z.B. MASKE1 EQ Mode). Als Ergebnis wird ein Byte oder Word zurückgegeben, das die Werte *true* (0FFFFH) oder *false* (0) enthält. Die Anweisung MOV AL, 4 EQ 3 lädt AL mit dem Wert 0, da der Ausdruck falsch ist.

### 4.5.8 Stringvergleiche

Ferner erlaubt der A86 Operatoren im Programm, die zur Übersetzungszeit einen Vergleich zweier Textstrings zulassen:

```
String1 EQ String2 (equal)
String1 NE String2 (not equal)
String1 = String2 (equal)
```

Um den Vergleich der beiden Strings korrekt abzuwickeln, müssen diese in die gleichen Delimiter (Trennzeichen) eingeschlossen werden. Das Ergebnis der Operation ist entweder *true* (0FFFFH) oder *false* (0). Mit dem Operator = werden zwei Zeichenketten verglichen, wobei die Zeichen mit 20H maskiert werden.

### 4.5.9 Definition von Datenbereichen

Bei der Definition der Größe von Variablen oder Konstanten werden die Operatoren BYTE, WORD, DWORD, QWORD und TWORD verwendet. Der A86 benötigt nur den ersten Buchstaben als Schlüsselwort (z.B. MOV [BX],B 30). Aus Gründen der Lesbarkeit und der Kompatibilität mit anderen Assemblern sollte allerdings die Langform des Schlüsselwortes verwendet werden.

### 4.5.10 SHORT/LONG/NEAR

Diese Schlüsselworte werden in Verbindung mit JMP-Anweisungen verwendet. Die Abkürzung LONG steht dabei für einen NEAR-Sprung im Assembler, d.h., es wird ein 3-Byte-Befehl generiert. Mit SHORT wird immer ein 2-Byte-Befehl ausgegeben. Wird einem Label das Zeichen '>' vorangestellt (z.B.: JMP >L1), handelt es sich um ein lokales Label, das bis zu 127 Byte entfernt stehen darf. In Abhängigkeit von SHORT oder LONG generiert der A86 dann einen 2- oder 3-Byte-Befehl. Aus Kompatibilitätsgründen empfehle ich aber, auf den Begriff LONG zu verzichten.

### 4.5.11 Der OFFSET-Operator

Beim Zugriff auf Speicherzellen (z.B.: MOV AX, Buffer) sind zwei Fälle zu unterscheiden. Einmal kann der Inhalt der Konstanten *Buffer* gemeint sein. Andererseits

besteht die Möglichkeit, die Adresse der Variablen *Buffer* in AX zu lesen. Die Anweisung:

```
MOV AX, Buffer
```

wird vom A86 deshalb in Abhängigkeit von der Deklaration von *Buffer* assembliert. Die Anweisung:

```
Buffer: DW ?
```

generiert das Label *Buffer:*, wodurch der Befehl:

```
MOV AX, Buffer
```

die Adreßkonstante (Offset) in AX lädt. Mit der Definition:

```
Buffer DW ?
```

(der `:` fehlt!) wird eine Word-Variable erzeugt. Mit:

```
MOV AX, Buffer
```

wird dann der Inhalt der Variablen (zu interpretieren als `MOV AX,[BUFFER]`) geladen. Um die Konstruktion etwas eindeutiger zu gestalten, existiert die Anweisung `OFFSET`, die in allen Beispielprogrammen verwendet wird. Mit:

```
MOV AX, OFFSET Buffer
```

wird eindeutig die (Offset-) Adresse der Variablen oder Konstanten *Buffer* in das Register AX geladen. Um den Inhalt der Variablen *Buffer* zu laden, ist die indirekte Befehlsform (z.B. `MOV AX,Buffer`) zu nutzen.

#### 4.5.12 Der TYPE-Operator

Mit diesem Befehl läßt sich während der Assemblierung der Typ eines Operators abfragen. Der Befehl liefert bei einer Bytevariablen den Wert 1 und bei einem Word den Wert 2 zurück. Der Operator läßt sich z.B. benutzen, um die Länge einer Datenstruktur in Byte zu ermitteln.

#### 4.5.13 Der THIS-Operator

Manchmal ist es erforderlich, den Wert des aktuellen Programmzeigers im Programm abzufragen. Hierfür unterstützt der A86 zwei Konstrukte. Entweder ist das Schlüsselwort `THIS` oder das Zeichen `$` einzusetzen. Die Anweisung:

Adresse1 EQU \$

weist dem Namen *Adresse1* zur Übersetzungszeit den Wert des aktuellen Programmzeigers zu. Die Konstante *Adresse1* kann dann in Ausdrücken verwendet werden.

Die bisher besprochenen Operanden lassen sich in Ausdrücken kombinieren. Dabei gelten die folgenden Prioritäten:

1. Klammern
2. Punkt
3. OFFSET, SEG, TYPE, PTR
4. HIGH, LOW, BIT
5. \*, /, MOD, SHR, SHL
6. +, -
7. EQ, NE, LT, LE, GT, GE, =
8. NOT, !
9. AND
10. OR, XOR
11. :, SHORT, LONG, BY
12. DUP

Die Zahl 1 steht dabei für die höchste Priorität.

## 4.6 Anweisungen zur Segmentierung

Ein übersetztes Programm besteht mindestens aus einem Code- und einem Datensegment. Bei der Erzeugung der .OBJ- und .COM-Files benötigt der A86 zusätzliche Informationen zur Generierung der Segmente (Reihenfolge, Lage, etc.). Nachfolgend werden die Steueranweisungen zur Definition der Segmente beschrieben.

### 4.6.1 CODE SEGMENT

Mit dieser Anweisung wird der A86 informiert, daß alle folgende Befehle im Codebereich zu assemblieren sind. Neben den Programmanweisungen können hier auch Konstanten oder initialisierte Daten abgelegt werden. Interessant ist dies vor allem bei der Erzeugung von .OBJ-Files, da der Linker die Segmente später zusammenfaßt. Das Ende des Codesegmentes sollte mit CODE ENDS abgeschlossen werden. A86 benötigt dies zwar nicht, es kommt aber der Lesbarkeit des Programmes zugute.

## 4.6.2 DATA SEGMENT

In diesem Segment werden Daten und Variablen abgelegt. Anweisungen wie DB, DW und STRUC erzeugen solche Variablen. Häufig findet man die Datendefinition zu Beginn eines Assemblerprogramms. Mit der ORG-Anweisung läßt sich die Lage des Datenbereiches explizit festlegen. In den Beispielprogrammen finden sich die Datenbereiche am Programmende hinter dem Code. Dadurch legt der Assembler automatisch die Lage des Datenbereiches korrekt fest. Code- und Datensegmente dürfen durchaus mehrfach im Programm definiert werden:

```
CODE SEGMENT
ORG 100

JMP NEAR Start

DATA SEGMENT
Txt DB "Hallo"
Val DB ?
X1 DW ?
DATA ENDS

CODE SEGMENT

Start: MOV AH, 09
      MOV DX, OFFSET Txt
      INT 21
      MOV AX, 4C00
      INT 21

      END
```

Das Datensegment sollte mit der Anweisung DATA ENDS abgeschlossen werden.

## 4.6.3 Die ORG-Directive

Bei OBJ-Dateien darf die Lage der Code- und Datenbereiche nicht festgelegt werden, da dies Aufgabe des Linkers ist. Bei COM- oder BIN-Programmen müssen dagegen die Adressen im Programm definiert werden. Hierzu dient die ORG-Anweisung, mit der sich die Adresse des nächsten Befehls definieren läßt. Mit ORG 100 wird A86 angewiesen, die nächste Anweisung ab CS:0100 zu assemblieren. Die Lage des Datensegments kann an den Codebereich angehängt werden. Dann entfällt die Adreßangabe für diesen Bereich. Der Parameter einer ORG-Anweisung muß entweder eine Konstante oder ein Ausdruck sein. Eine Verwendung eines symbolischen Namens ist ebenfalls möglich, sofern keine Vorwärtsreferenzen dabei notwendig werden. Wird die Anweisung ORG 0 verwendet, legt A86 eine BIN-Datei an, das Programm beginnt ab der Offsetadresse 0. Bei der Assemblierung einer COM-Datei generiert der A86 den Code auch ohne die ORG 100 Anweisung ab CS:0100. In den Beispielprogrammen wird die ORG-Anweisung jedoch verwendet. Allgemein bleibt festzuhalten, daß der ORG-Befehl nur selten im Programm auftreten sollte.

Andernfalls besteht die Gefahr, daß sich Codebereiche bei späteren Programmiererweiterungen überlappen und damit zu Fehlern führen.

#### 4.6.4 Die EVEN-Directive

Mit dieser Anweisung erzwingt der Programmierer, daß der nächste assemblierte Befehl oder die Lage einer Variablen auf einer geraden Speicheradresse festgelegt wird. Die Anweisung:

```
ORG EVEN $
```

legt den Beginn der nächsten Anweisung auf eine gerade Adresse. Innerhalb des Codesegments fügt der A86 gegebenenfalls eine NOP-Anweisung ein. Variable werden einfach auf eine gerade Adresse gelegt. Die EVEN-Directive ist besonders bei 80X86-Prozessoren hilfreich, da Zugriffe auf gerade Adressen schneller ausgeführt werden. Ein 16-Bit-Zugriff auf eine ungerade Adresse erfordert 2 Speicheroperationen.

### 4.7 Datendefinitionen mit DB, DW, DD, DQ, DT

Zur Definition von Variablen kennt der A86 - wie MASM und TASM - die Pseudoanweisungen:

```
DB (erzeugt ein Data Byte)
DW (erzeugt ein Data Word)
DD (erzeugt ein Double Data Word = 4 Byte)
DQ (erzeugt ein Quad Data Word = 8 Byte)
TB (erzeugt 10 Data Bytes)
```

Die DQ-Anweisung erzeugt 8 Byte, die zur Speicherung von 8087-Werten gebraucht werden. Die TB-Anweisung erzeugt eine Variable mit 10 Byte. Vor dem Pseudobefehl kann der Name der Variablen stehen:

```
Text DB 'Dies ist ein Text'
Zahlen DB 1,2,3,4,5,6
Wort DW 0FFFFH
```

Über den Namen läßt sich später auf die Daten zurückgreifen (z.B. MOV DX,OFFSET Text). Dabei ist ein Sonderfall zu beachten: steht hinter dem Namen ein Doppelpunkt, legt der A86 ein Label an. Ein MOV DX, Text liest dann die OFFSET-Adresse direkt in AX. Aus Gründen der Übersichtlichkeit sollte hierauf jedoch verzichtet werden.

Solange die Konstanten mit Werten initialisiert werden, müssen sie im Codesegment abgelegt werden. Um größere Datenbereiche mit dem gleichen Wert zu initialisieren, existiert das Schlüsselwort `DUP`. Mit den Anweisungen:

```
CODE SEGMENT
Table:
DB 100 DUP 0
```

wird ab dem Label *Table* ein Datenbereich von 100H Byte reserviert und mit dem Wert 0 initialisiert. Die Anweisungen lassen sich dabei auch zu Sequenzen zusammenfassen. Der Befehl:

```
DW 2 DUP (0, 5 DUP 3)
```

ist äquivalent dem Befehl:

```
DW 0,3,3,3,3,3,0,3,3,3,3,3
```

Werden die Daten als Variable im Datensegment abgelegt, ist eine Initialisierung nicht sinnvoll, da diese nicht mit geladen wird. Deshalb kann dann die Wertangabe entfallen - statt dessen schreibt man ein Fragezeichen. Eine Variable *Buffer* mit 256 Byte läßt sich dann mit:

```
Buffer DW 100 DUP ?
```

anlegen. Die Anweisung `DUP ?` steht als Platzhalter für die 256 Datenbytes im Segment. Wichtig ist, daß die Definition im Datensegment steht. Bei OBJ-Dateien faßt der A86 alle Definitionen im korrekten Datensegment zusammen. Das Fragezeichen veranlaßt in COM-Dateien die Initialisierung der Daten mit 0. ASCII-Strings lassen sich einfach in Hochkommas einschließen und im Anschluß an eine DB-Anweisung eingeben:

```
DB 'Dies ist ein Text',0D,0A
```

Zur Wahrung der Kompatibilität dürfen für Textkonstanten auch Hochkommata verwendet werden. Es ist aber auf die unterschiedliche Speicherung von Bytes und Worten zu achten. Die Anweisungen:

```
DB 'AB'   (speichert LOW 41H, High 42H)
DW 'AB'   (speichert 42H, 41H)
```

legen die Bytes in unterschiedlichen Speicherzellen ab. Im ersten Fall steht der Wert 'A' auf dem unteren Byte, daran schließt sich der Buchstabe 'B' an. Bei der Abspeicherung von Worten ist der linke Buchstabe dem höheren Byte zugeordnet. Mit der Anweisung:

```
DD 0FFFF:0000
```

lassen sich 32-Bit-Werte (FAR Pointer) definieren. Das High und Low Word wird durch einen Doppelpunkt getrennt.

## 4.8 Die Definition von Strukturen

Mit der Anweisung `STRUC` lassen sich mehrere Variablen oder Konstanten zu kompakteren Datenstrukturen zusammenfassen. Die Directive erzwingt, daß die Variablen zusammen abgespeichert werden und sich anschließend durch die Zeigerregister `BX`, `BP`, `DI` oder `SI` über das erste Element adressieren lassen. Die indirekte Adressierung über `[BX+SI+Konst.]` eignet sich gut zur Bearbeitung solcher Strukturen. Die Definition:

```
PSP STRUC [BX]
    Dummy DB 080 DUP    ; Länge Vorspann
    Count DB ?          ; Zeichenzähler
    Buff  DB 0FF DUP ?  ; Puffer
ENDS
```

schaltet den A86 in einen mit dem `DATA SEGMENT`-Mode vergleichbaren Zustand und definiert den in unserem Beispiel `ASK.ASM` benutzten Datenpuffer. Auf die Variable `Count` der Struktur läßt sich dann mit:

```
MOV CL, Count ; [BX + 80]
```

direkt zugreifen, der A86 generiert dann den korrekten Index, der sich auf `BX` bezieht. Nach der `STRUC`-Anweisung steht das Indexregister `BX`, `BP`, `SI` oder `DI`, das zur Adressierung verwendet werden soll. Die Struktur muß mit den Schlüsselwort `ENDS` abgeschlossen werden.

## 4.9 Vorwärtsreferenzen

Der A86 erlaubt es, daß sich verschiedene Programmelemente (`JMP`, `CALL`, etc.) auf Labels beziehen, die erst im nachfolgenden Programmteil definiert werden (Vorwärtsreferenzen). Folgender Sprung verdeutlicht diesen Sachverhalt:

```
        JNZ Test2
        .
        .
Test2:  CMP AL,03F
        .
```

Bei bedingten Sprüngen wird dann immer ein 2-Byte-Befehl kodiert. Beim `JMP`-Befehl hängt die generierte Codelänge von der Entfernung ab. Sprungweiten über 127

Byte sind als JMP NEAR zu kodieren, andernfalls genügt ein JMP SHORT. Der Assembler kann hier nicht automatisch den korrekten Wert einsetzen, da er noch nicht die Entfernung kennt. Daher wird bei einem JMP xxxx auf eine Vorwärtsreferenz immer der 3-Byte-JMP-Befehl verwendet. Mit einem Bezug auf ein lokales Label läßt sich aber ein 2-Byte-Befehl (z.B. JMP >L1) erzwingen. Diese Regel wird aber durch die Schlüsselworte NEAR oder SHORT überschrieben.

Zudem kann im Programm auf Variablen Bezug genommen werden, die erst im Verlauf des Programmes definiert werden (z.B. das Datensegment liegt am Programmende). Dies war bei älteren Versionen von A86 nicht der Fall, weshalb in vielen Programmen noch die Daten im Programmkopf zu definieren sind. Bei Zugriffen auf Daten sollte der Datentyp aber explizit angegeben werden.

Bei Ausdrücken besteht die Problematik, daß Operanden an der zu assemblierenden Stelle noch unbekannt sind, falls eine Vorwärtsreferenz Verwendung findet. Der A86 kann dies nicht auflösen, mit einem Trick läßt sich aber die Berechnung aber dennoch durchführen. Sie ist einfach an die Programmstelle zu verlagern, an der der Operator definiert wird. Die folgende Sequenz bestimmt den Wert des Segmentregisters CS am Programmende + 15.

```
MOV AX,CS          ; lade aktuellen Wert
ADD AX, SEG_SIZE  ; addiere Codelänge hinzu
.
.
.
PROG_SIZE EQU $    ; Programmzeiger am Programmende !
SEG_SIZE EQU (PROG_SIZE+15)/16 ; nächste Segmentadr.
```

Zur Sicherheit wird auf die letzte Programmadresse (angegeben durch \$) der Offset von 15 addiert. Damit liegt der Offsetwert im nächsten Paragraphen (Adresse/16). Durch Division mit 16 läßt sich der korrekte Segmentwert als Paragraph ermitteln.

## 4.10 Die EQU-Directive

Diese Directive wurde bereits in mehreren Beispielen verwendet. Mit EQU läßt sich einem Namen eine Konstante, ein Ausdruck, etc. zuweisen. Immer wenn im Verlauf des Programmes dann dieser Name auftaucht, ersetzt der A86 diesen Namen durch den mit EQU zugewiesenen Wert. Statt der Anweisung: MOV AX,0FFFF läßt sich das Programm wesentlich transparenter mit der Sequenz:

```
True EQU 0FFFF      ; Konstante True
False EQU 0         ; Konstante False
.
.
MOV AX,True         ; AX Init
```

gestalten. Ähnlich lassen sich Masken, Pufferlängen, etc. mit symbolischen Namen belegen und per EQU definieren. Im Programm taucht dann nur noch der symbolische

Name auf. Werden die EQU-Anweisungen im Programmkopf definiert, lassen sich die Programme sehr transparent gestalten, da sich diese EQUs leicht ändern lassen.

Zusätzlich sind auch Ausdrücke mit EQU definierbar. Die Anweisung:

```
MAX_NAME EQU [BX+1]
```

läßt sich in weiteren Befehlen sehr leicht verwenden. Der Befehl:

```
MOV CX, MAX_NAME
```

wird dann vom A86 durch `MOV CX,[BX+1]` ersetzt. Mit der EQU-Anweisung lassen sich auch Symbole für andere Assembler generieren, die im Programm verwendet werden dürfen. Manchmal kommt eine Variable im Codesegment, ein andermal im Stacksegment und im dritten Fall im Datensegment vor. Probleme bereitet dieser Fall, wenn mit Segment-Override-Befehlen auf die Variable zugegriffen werden muß. Mit der Anweisung:

```
QS EQU CS
```

wird QS als Override für CS definiert. Alle Referenzen:

```
QS: MOV AX,[BP+10]
```

werden dann über CS ausgeführt. Wird QS mit einem EQU auf DS gesetzt, beziehen sich alle Overrides auf das Datensegment. Die Lage des Variablensegmentes läßt sich an einer zentralen Stelle definieren und ist damit leicht änderbar. Jede Override-Anweisung erzeugt ein Opcodebyte im Programm. Paßt das Programm in ein Segment (z.B. COM Programm benutzt 64 KByte für CODE, DATA und STACK), ist ein Segment-Override nicht notwendig. Mit der zwangsweisen Definition von:

```
QS EQU CS
```

verlängert sich das Programm bei jedem Aufruf von QS um ein Byte und die Laufzeit wird auch verschlechtert. Um die unnötige Codegenerierung durch den QS-Befehl zu vermeiden, kann in der EQU-Definition der Operator NIL verwendet werden. NIL veranlaßt, daß der Assembler den symbolischen Namen aus dem Quellcode entfernt. Dadurch wird für den Override-Befehl QS kein Code generiert.

Ferner lassen sich die EQU-Definition verwenden, um einen Interrupt mit einem symbolischen Namen zu versehen. Der INT 3 wird häufig von Debuggern zum Test verwendet. Die CPU führt im Single-Step-Mode nach jedem Befehl einen INT 3 aus. Es besteht daher die Möglichkeit, den INT 3 durch:

```
TRAP EQU INT 3
```

umzudefinieren. Immer wenn im Programm der Begriff TRAP auftritt, ersetzt der A86 diesen durch die INT 3-Anweisung.

Mit EQU-Anweisungen kann man in den meisten Assemblern einem Namen durchaus mehrfach verschiedene Werte innerhalb des Programmes zuweisen (siehe A86-Dokumentation). Im A86 sind Mehrfachdeklarationen mit EQU unzulässig. Mit dem aus Kompatibilitätsgründen eingeführten Operator = kann diese Restriktion allerdings umgangen werden. Sie ist als Synonym für EQU zu verwenden. Soll der Name eindeutig und nicht änderbar sein, ist aber darauf zu achten, daß die erste Definition des Namens durch ein EQU erfolgt. Wird der Name erstmals mit dem Gleichheitszeichen definiert, legt der A86 ein lokales Symbol an. Dann kann die Definition beliebig neu belegt werden. Mit dieser Konstruktion lassen sich Assemblervariablen für bedingte Übersetzung etc. erzeugen. Mit dem Operator = kann diese Restriktion allerdings umgangen werden. Sie ist als Synonym für EQU zu verwenden. Soll der Name eindeutig und nicht änderbar sein, ist aber darauf zu achten, daß die erste Definition des Namens durch ein EQU erfolgt. Wird der Name erstmals mit dem Gleichheitszeichen definiert, legt der A86 ein lokales Symbol an. Dann kann die Definition beliebig neu belegt werden. Mit dieser Konstruktion lassen sich Assemblervariablen für bedingte Übersetzung etc. erzeugen

## 4.11 Die PROC-Directive

Diese Anweisung wurde aus Kompatibilitätsgründen in den A86 mit aufgenommen. Sie muß bei MASM am Beginn eines Unterprogrammes (Prozedur) stehen. Dabei sind die Schlüsselworte:

```
Name PROC FAR  
Name PROC NEAR  
Name PROC
```

anzugeben. Name steht dabei für den Namen der Prozedur, während die Schlüsselworte FAR und NEAR die Aufrufform für den CALL-Befehl festlegen. Sobald ein RET-Befehl auftritt, ersetzt der Assembler diesen durch die betreffende RET- oder RETF-Anweisung. Der Programmierer braucht also nur einen Befehlstyp für RET einzusetzen und der Assembler ergänzt den Befehl. Viele Programmierer ziehen aber vor, den RET-Befehl explizit (notfalls auch als RETF) zu schreiben. Dies ist bei der Wartung älterer Programme hilfreich, da sofort erkennbar wird, ob eine Prozedur mit FAR oder NEAR aufzurufen ist. Wird eine Prozedur mit PROC definiert, muß am Ende des Programmes das Schlüsselwort ENDP stehen.

## 4.12 Die LABEL-Directive

Dieser Pseudobefehl wurde ebenfalls aus Kompatibilitätsgründen zu anderen Assemblern eingeführt. Die Directive besitzt das Format:

```
Name LABEL NEAR
Name LABEL FAR
Name LABEL BYTE
Name LABEL WORD
```

und definiert eine Marke für einen Sprung. In den Beispielprogrammen wird diese Directive nicht benutzt. Die Anweisung *Name LABEL NEAR* läßt sich in den Programmen wesentlich einfacher mit *Name:* ersetzen. Hinter dem Labelnamen wird einfach ein Doppelpunkt (:) angegeben. Dadurch kann der Punkt innerhalb des Segments angesprungen werden. Die Notation wird auch durch andere Assembler unterstützt. Die Bezeichnungen *Name LABEL BYTE* und *Name LABEL WORD* lassen sich ebenfalls durch die allgemein akzeptierten Schlüsselworte DB und DW ersetzen.

Lediglich die Konstruktion *Name LABEL FAR* fällt etwas aus dem Rahmen, da sie von anderen Assemblern meist nicht unterstützt wird. Sie definiert eine Marke, die sich aus externen Codesegmenten per CALL FAR aufrufen läßt. Damit muß die Prozedur, die durch das Label identifiziert wird, mittels eines RETF beendet werden. Der A86 besitzt in diesem Zusammenhang noch eine weitere Eigenart: Beim CALL-Aufruf einer mit LABEL FAR definierten Prozedur generiert der Assembler eine Sequenz :

```
PUSH CS
CALL NEAR Labelname
```

Diese Konstruktion besitzt den gleichen Effekt wie ein 5-Byte-CALL FAR, belegt aber nur 4 Byte. Ein CALL-Aufruf auf ein LABEL FAR darf nicht auf Vorwärtsreferenzen angewandt werden, da sonst nur ein CALL NEAR ausgeführt wird, was hier zu einem Fehler führt. Die LABEL-Definition muß daher immer vor dem CALL-Aufruf stehen. Aus meiner Sicht dürfte die Benutzung dieser Konstruktion sehr selten erforderlich sein und aus Gründen der Kompatibilität sollte sie komplett ignoriert werden.

Die nachfolgenden Directiven beziehen sich auf getrennt zu assemblierende Module und sind für den Linker erforderlich.

## 4.13 Die NAME-Directive

Diese Directive definiert einen Namen für das folgende Modul. Dieser Name wird durch den Linker ausgewertet, um Referenzen aus anderen Programmen auf den

betreffenden Modulnamen aufzulösen. Die Directive darf aber an jeder Stelle innerhalb des Programmes stehen. Fehlt die NAME-Anweisung, setzt A86 einfach den Namen des .OBJ-Files - ohne OBJ-Extension - als Modulname ein. Enthält ein Programm mehr als eine NAME-Directive, übernimmt der A86 nur die letzte gültige Anweisung.

## 4.14 Die PUBLIC-Directive

Diese Anweisung erlaubt die explizite Auflistung von Symbolen (Variablen, Prozeduren, etc.), die durch andere Module adressierbar sein sollen. Der Linker wertet diese Informationen aus den .OBJ-Dateien aus und verknüpft offene Verweise auf diese Symbole mit den entsprechenden Adressen. So kann zu einem Programm eine Prozedur aus einer fremden Objektdatei zugebunden werden. Der Linker setzt dann im CALL-Aufruf die Adresse des Unterprogrammes ein. Ferner lassen sich mit der PUBLIC-Directive Variablen als global definieren, so daß andere Module darauf zugreifen können. Fehlt die PUBLIC-Anweisung in den Programmen, nimmt der A86 alle globalen Labels und Variablennamen als PUBLIC an. Lokale Symbole in EQU-Anweisungen, lokale Labels (1 Buchstabe, gefolgt von mehreren Ziffern, z.B. L1), Variable in Strukturen und im DATA Segment werden allerdings nicht automatisch als PUBLIC erklärt. Die Anweisung PUBLIC ohne einen Namen schaltet den *explizit Mode* ein, d.h., die Symbole werden nicht automatisch als PUBLIC erklärt. Die Anweisung sollte im Modulkopf stehen und kann folgende Form besitzen:

PUBLIC Multi, Basis, Result

Die Symbole Multi, Basis und Result lassen sich dann aus anderen Modulen ansprechen, wenn sie dort als EXTRN erklärt werden. Fehlt die PUBLIC-Anweisung, wird der Linker eine Fehlermeldung mit den offenen Referenzen angeben. Zur Erhöhung der Programmentransparenz und zur Vermeidung von Seiteneffekten sollten nur die globalen Symbole in der PUBLIC-Anweisungsliste aufgeführt werden. Wird das Symbol im folgenden Programmcode des Moduls nicht definiert, generiert der A86 automatisch eine Fehlermeldung mit dem Hinweis auf das Fehlen des Symbols.

## 4.15 Die EXTRN-Directive

Diese Directive ist das Gegenstück zur PUBLIC-Directive. Soll in einem Programm auf ein Unterprogramm oder eine Variable aus einem anderen .OBJ-File zugegriffen werden, muß dies dem A86 bekannt sein. Mit der EXTRN-Directive nimmt der A86 an, daß das Symbol in einer externen .OBJ-Datei abgelegt wurde und durch den Linker überprüft wird. Ohne EXTRN-Directive gibt der A86 eine Fehlermeldung aus, da das Symbol im folgenden Code nicht definiert wird. Die EXTRN-Directive besitzt folgendes Format:

```
EXTRN Name1:Typ, Name2:Typ, ....
```

Als Name ist der entsprechende Symbolname einzutragen. Ferner muß der Typ der Referenz in der Deklaration angegeben werden. Hierfür gilt:

```
B oder BYTE oder NEAR  
W oder WORD oder ABS  
D oder DWORD  
Q oder QWORD  
T oder BYTE  
F oder FAR
```

Das Gegenstück zu obiger PUBLIC-Definition ist z.B. die Erklärung der externen Referenzen im Hauptprogramm:

```
EXTRN Multi:FAR, Basis:WORD, Result:DWORD
```

Tritt nun eine Referenz auf ein solches Symbol auf (z.B.: MOV AX, Basis), generiert der A86 einen indirekten Zugriff (MOV AX, [Basis]). Ohne die EXTRN-Definition wird mit MOV AX, Basis die Konstante Basis in AX geladen. Fehlt die Definition dieser Konstanten im Programm würde A86 eine Fehlermeldung erzeugen.

Der A86 besitzt allerdings bei der Auflösung externer Referenzen einen meiner Ansicht nach recht unangenehmen Nebeneffekt. Alle Referenzen, die am Ende eines Übersetzungslaufes noch offen sind, werden automatisch als EXTRN deklariert. Es empfiehlt sich deshalb, sehr gewissenhaft die PUBLIC- und EXTRN-Deklarationen zu benutzen und sich nicht auf die automatische Vergabe durch den A86 zu verlassen. Mit der Option +x wird übrigens eine explizite Definition erzwungen.

## 4.16 Die MAIN-Directive

Wird ein lauffähiges Programm aus mehreren .OBJ-Dateien gelinkt, muß ein Modul als Hauptprogramm fungieren. Hierfür existiert die Directive MAIN. Das Modul mit diesem Schlüsselwort enthält die Programmstartadresse. Die Anweisung:

```
MAIN:
```

wird am Anfang des Hauptprogrammes angegeben, womit die Startadresse definiert ist.

**Anmerkung:** Der Wert der Startadresse wird im allgemeinen aber durch das Betriebssystem gesetzt.

## 4.17 Die END-Directive

Diese Directive wurde aus Kompatibilitätsgründen zu anderen Assemblern eingeführt. Mit dieser Anweisung wird das Ende des Assemblermoduls markiert. Die Anweisung besitzt die Form:

```
END  
END start_adr
```

wobei der Name *start\_adr* als Label für Referenzen dienen darf. A86 behandelt den hinter END angegebenen Namen wie ein Symbol, das mit EQU definiert wurde. So läßt sich dann die Programmlänge leicht ermitteln. Folgen weitere Anweisungen, werden diese als getrenntes Programm übersetzt. Damit lassen sich in einer Quellcodedatei mehrere Module ablegen und mit einem Durchlauf assemblieren (s. ESC.ASM).

## 4.18 Die SEGMENT-Directive

Die Anweisung signalisiert dem A86, wie der assemblierte Objectcode abzulegen ist. Das Format der Directive ist wie folgt definiert:

```
Seg-Name SEGMENT [align] [combine] ['class_name']
```

Die in Klammern [] stehenden Parameter sind optional. Der angegebene Seg-Name wird durch den Linker bearbeitet. Damit läßt sich direkt im Programm auf dieses Symbol zugreifen, um zum Beispiel den Wert des Segmentes zu laden. Die Anweisungen:

```
Data_Seg SEGMENT  
.  
.  
Start: MOV AX, Data_Seg  
      MOV DS, AX
```

sorgen dafür, daß DS mit der korrekten Segmentadresse geladen wird. Der Linker ersetzt den Namen *Data\_Seg* durch die korrekte Adresse.

Mit dem Parameter *align* läßt sich spezifizieren, wie die Segmente angelegt werden. Es sind die Schlüsselworte: BYTE, WORD, PARA und PAGE erlaubt. Mit BYTE wird der Codebeginn auf eine Bytegrenze fixiert. WORD erzwingt, daß das Segment auf geraden Adressen beginnt. Mit PARA wird die Segmentgrenze auf eine durch 16 teilbare Adresse (Paragraph) festgelegt. PAGE justiert den Beginn eines Segments auf eine durch 256 teilbare Adresse. Ein Datenbereich läßt sich dann z.B. mit Daten SEGMENT WORD anlegen. Damit ist sichergestellt, daß die erste Variable an einer

geraden Adresse liegt. Fehlt die `Align-Option`, richtet der A86 die Segmente automatisch auf Paragraphen aus.

Wird der Segmentname als Teil einer `GROUP`-Deklaration benutzt, dann legt A86 den Code in das betreffende Segment der Gruppe ab. In einem Programm dürfen beliebig viele Segmente definiert werden. Wird ein Name innerhalb eines Programmes mehrfach einem Segment zugewiesen, assembliert der A86 den Code in einen gemeinsamen Block.

Mit dem optionalen Parameter *combine* wird festgelegt, wie Codebereiche mit gleichem Namen aus verschiedenen Modulen zu kombinieren sind. Der A86 erlaubt Optionen für den Parameter `PUBLIC`, `STACK`, `COMMON`, `MEMORY` und `AT Nummer`. Wird `AT Nummer` verwendet, muß ein Komma folgen, falls sich noch ein weiterer Parameter anschließt. Für die Parameter gelten folgende Regeln:

- ◆ Mit `PUBLIC` werden alle Programmmodule aus verschiedenen `OBJ`-Files in einem Segment aneinander gefügt. Die Größe des Segmentes bestimmt sich aus den Modulgrößen plus den eventuell durch `align` entstehenden Lücken.
- ◆ Mit `STACK` wird ein Segment für den Systemstack angelegt. Alle Stacksegmente innerhalb der Module werden in diesem Bereich gesammelt. Ein Stacksegment läßt sich in einem Programm mit folgender Konstruktion definieren:

```
STACK SEGMENT WORD STACK
DW 0100 DUP (?)
TOP_OF_STACK :
```

Damit werden 256 Worte als Stackbereich reserviert und die oberste Adresse wird durch `TOP_OF_STACK` markiert. Treten nun solche Deklarationen in mehreren Modulen auf, addiert der Linker die Werte zu einem gemeinsamen Bereich.

- ◆ `COMMON` stammt aus dem `FORTRAN`-Bereich, wird aber durch den `MS-Linker` nicht unterstützt. Dieser Parameter sollte deshalb nicht verwendet werden.
- ◆ Mit *AT Nummer* läßt sich ein Segment auf eine absolute Adresse legen. Die Adresse wird durch den Parameter *Nummer* definiert. Mit diesem Parameter läßt sich zum Beispiel die `Interrupt-Tabelle` recht einfach definieren:

```
Vec SEGMENT AT 0
```

Fehlen die *combine*-Parameter in Segmenten mit Namensangaben, wird das Segment mit keinem anderen Segment kombiniert.

Als letzter Parameter läßt sich der *class name* - in Hochkommata - angeben. Damit werden einzelne Programmteile einer Klasse (`CODE`, `DATA`) zugeordnet. Dies ist bei Verwendung von Hochsprachen interessant, die je nach Speichermodell bestimmte Klassen verwenden. Näheres findet sich in den entsprechenden Handbüchern der Compiler.

## 4.19 CODE-, DATA- und STACK-Directiven

Diese Deklarationen wurden in den A86 aufgenommen, um eine Kompatibilität mit Turbo Pascal und MASM herzustellen. Im DATA Segment werden lediglich Speicherbereiche reserviert, aber kein Code und keine Initialisierungswerte abgelegt. Das gleiche gilt für das STACK-Segment. Erst mit Auftreten eines CODE-Segments beginnt die Generierung von Objectcode durch A86.

## 4.20 Die ENDS-Directive

Mit dieser Directive wird die Deklaration eines Segmentes aufgehoben. Damit lassen sich verschiedene Segmente innerhalb eines Programmes schachteln. Die Definition:

```
_DATA SEGMENT BYTE PUBLIC 'DATA'
    LEN DB ?
    BUFF DB 255 DUP (?)
_DATA ENDS
```

legt ein Datensegment an, das über den Namen `_DATA` angesprochen werden kann.

Bei einigen Assemblern müssen alle Anweisungen innerhalb solcher SEGMENT-Anweisungen stehen. Der A86 erlaubt aber die Assemblierung auch ohne solche Anweisungen. Der erzeugte Code wird in das erste auftretende Segment mit dem Namen `_TEXT` abgelegt. Existiert kein solches Segment, erzeugt der A86 ein eigenes Segment zur Ablage des Programmes. Wird das SMALL-Modell benutzt, d.h., im Code kommt kein RETF vor, wird das Segment mit:

```
_TEXT SEGMENT PUBLIC BYTE 'CODE'
```

vereinbart. Falls FAR-Aufrufe auftreten, generiert der A86 das Segment:

```
Name_TEXT SEGMENT PUBLIC BYTE 'CODE'
```

An Stelle von Name wird der Name des Moduls eingetragen. Dabei wird entweder die NAME-Directive ausgewertet oder der Name des OBJ-Files übernommen.

## 4.21 Die GROUP-Directive

Mit dieser Directive wird der Linker angewiesen, alle angegebenen Programmsegmente innerhalb eines 64-KByte-Blocks zu kombinieren. Es gilt dabei die Syntax.

```
Group_name GROUP Seg_name1, Seg_name2, ...
```

Der *Group\_name* läßt sich dann innerhalb der Module der Gruppe verwenden (z.B.: MOV AX,Group\_name). Passen die Module nicht in einen 64-KByte-Block, gibt der Linker eine Fehlermeldung aus. Mit dieser Anweisung lassen sich einzelne Module zu Gruppen kombinieren und in einem Segment ablegen. Zu Beginn des Segmentes werden die Segmentregister gesetzt und sind für alle Module der Gruppe gültig. Wird die GROUP-Directive verwendet, muß sie allerdings in allen Modulen benutzt werden, da sonst einzelne Module nicht im Block kombiniert werden.

## 4.22 Die SEG-Directive

Diese Directive wurde aus Kompatibilitätsgründen zu anderen Assemblern und zu Borland (Turbo) C aufgenommen. Der Directive kann eine Variable oder ein Label zugeordnet werden, die in einem eigenen Segment abgelegt wird. Dann kann LINK die Referenzen auf dieses Segment berechnen und nachträglich auflösen. Die Directive darf nur bei der Assemblierung in OBJ-Files verwendet werden.

## 4.23 Makros und bedingte Assemblierung

Der A86-Assembler besitzt die Fähigkeit zur Verarbeitung von Textmakros. Als Makro wird hierbei eine Folge von Anweisungen verstanden, die unter einem (Makro-) Namen zusammengefaßt werden. Im Programm reicht es dann, lediglich den Makronamen einzusetzen. Wird ein solcher Makroname gefunden, durchsucht der Makroprozessor alle Makrodefinitionen und tauscht den Makronamen im Quelltext gegen die Befehle des Makros aus. Dadurch lassen sich die Quellprogramme mit recht mächtigen Befehlen ausstatten. Ich unterscheide beim A86 zwischen impliziten und expliziten Makros. Der A86 besitzt einige Makros, die fest (implizit) im Assembler implementiert sind. Bei den expliziten Makros muß der Programmierer die Definition selbst vornehmen.

Ferner bietet der A86 die Möglichkeit der bedingten Assemblierung, d.h. Teile des Quellcodes werden nur in Abhängigkeit von einer Bedingung übersetzt. In den nachfolgenden Abschnitten möchte ich diese Eigenschaften des A86 vorstellen.

### 4.23.1 Die impliziten Makros des A86

Der A86 kennt einige Anweisungen, die nicht im 8086-Befehlsvorrat enthalten sind. Tritt eine solche Anweisung auf, generiert der Assembler eine Folge von 8086-Befehlen.

## Das IF-Statement

Häufig kommt es vor, daß innerhalb eines Assemblerprogramms ein Befehl, in Abhängigkeit von einer Bedingung, übersprungen werden soll. Im A86-Befehlssatz läßt sich dies zum Beispiel mit folgender Sequenz erledigen:

```
JNZ >L1      ; JMP lokales Label L1 falls <> 0
MOV AX,BX    ; Folgebefehl
L1: ....     ; Sprungziel
```

Es wird hier die Bedingung *IF Not Zero* abgefragt. Ist die Bedingung erfüllt, wird der nachfolgende Befehl übergangen. Die Konstruktion ist besonders für Einsteiger recht unübersichtlich. Der A86 bietet deshalb ein internes Makro:

```
IF Z MOV AX,BX
```

der den gleichen Code wie die obige Sequenz generiert. Im Programm steht aber nur die eine Anweisungszeile, die anschaulich beschreibt, was der Befehl leistet. Als Bedingung innerhalb des IF-Statements darf jede Konstruktion der bedingten Sprungbefehle (Z, NZ, G, GE, C, NC, etc.) stehen. Der A86 setzt dann die invertierte Form ein und generiert das J-Zeichen als Präfix. Eine Ausnahme bildet nur die Bedingung JCXZ, die keine äquivalente negierte Form besitzt und daher nicht verwendet werden darf.

## Mehrfach PUSH-, POP-, INC- und DEC-Befehle

Die 8086-Befehle lassen immer nur ein PUSH oder POP zu. Sollen nun die Register AX und BX auf dem Stack gesichert werden, erfordert dies die Befehle:

```
PUSH AX
PUSH BX
```

Hier bietet der A86 wieder einen eingebauten Makro, so daß sich die obige Sequenz recht einfach durch:

```
PUSH AX, BX
```

nachbilden läßt. Der A86 generiert dann die zwei PUSH-Befehle. Das gleiche gilt für die POP-Anweisung, so daß zum Beispiel die Konstruktion POP AX, BX, DX, CX die vier Register vom Stack restauriert.

Auch bei den INC- und DEC-Befehlen ist es oft wünschenswert, in der Anweisungszeile die Zahl der Increment-/Decrement-Schritte angeben zu können. Mit A86 steht dem nichts mehr im Wege:

```
INC AX,4
```

sorgt zum Beispiel dafür, daß im Quellprogramm der Befehl `INC AX` viermal hintereinander übersetzt wird. Dabei ist sogar noch eine Steigerung möglich. Mit:

```
INC AX, BL, 2
```

wird die Sequenz:

```
INC AX
INC BL
INC AX
INC BL
```

erzeugt. Bei Speicheroperanden (z.B. `INC LEN, 3`) funktioniert das Makro nur, falls es sich um keine Vorwärtsreferenz handelt, d.h., der Name muß bereits definiert sein. Als Alternative kann in obigem Fall der Additionsbefehl verwendet werden (`ADD LEN, 2`).

## Bedingte RETURN-Anweisungen

Als Programmierer vermißt man häufig innerhalb eines Unterprogramms bedingte `RET`-Anweisungen (wie beim 8080/Z80). Der 8086-Befehlssatz bietet hier nichts, weshalb man auf die Konstruktion:

```
JNZ >L1 ; übergehe den nächsten Befehl
RET     ; Rücksprung
L1: ...
```

ausweichen muß. Mit A86 lassen sich bedingte `RET`-Anweisungen als implizite Makros formulieren:

```
RZ RET
RNZ RETF
RC IRET
```

Die obigen Beispiele zeigen die Anwendung auf die drei möglichen `RET`-Anweisungen. Mit `RZ RET` wird das Programm beendet, falls das Zero-Flag gesetzt ist. Der A86 arbeitet allerdings bei der Umsetzung des Makros mit einem recht undurchsichtigen Trick. Es wird kein `RET`-Befehl im Code eingesetzt, sondern der Sprung bezieht sich auf eine `RET`-Anweisung innerhalb des Unterprogrammes. Da der bedingte Sprung nur 127-Byte übergehen kann, muß ein `RET` innerhalb dieser Distanz im Code vorliegen. Fehlt dieses `RET`, generiert der A86 eine *02 Jump > 128*-Meldung. Ein Ausweg ist dann nur noch, innerhalb der geforderten Distanz ein `RET`-Statement im Code einzusetzen. Falls keine geeignete Stelle im Code existiert, kann der bereits besprochene `IF`-Befehl (z.B. `IF Z RET`) eingesetzt werden, der einen expliziten `RET`-Befehl einsetzt.

## Extended MOV- und XCHG-Anweisungen

Der A86 besitzt zudem einige eingebaute Makrofunktionen zur Erweiterung der MOV- und XCHG-Befehle. Eine der Einschränkungen des 8086-Befehlssatzes ist es, daß ein Segmentregister nicht direkt mit einer Konstanten geladen werden darf. Die Anweisung:

```
MOV DS,03FFF
```

wird vom Assembler normalerweise mit einer Fehlermeldung quittiert. Das gleiche gilt für die Anweisung:

```
MOV DS,ES
```

Beim A86 ist dies anders, hier wird für MOV DS,ES die Sequenz:

```
PUSH ES  
POP DS
```

generiert. Bei Verwendung der Konstruktion MOV DS,03FFF erzeugt der Assembler die Sequenz:

```
PUSH AX  
MOV AX,03FFF  
MOV DS,AX  
POP AX
```

Weiterhin erlaubt der A86 3 Operanden für den MOV-Befehl (z.B. MOV AX,BX,03FFF). Dieser Befehl wird in die Sequenz:

```
MOV BX,03FFF  
MOV AX,BX
```

umgesetzt. Die Konstruktion erlaubt allerdings kein Segment-Override, da dieses für jeden Befehl gelten würde. Ferner ist es möglich, mit dem A86-Kommando: MOV [3000],[BX+10] einen direkten MOV-Befehl zwischen zwei Speicherzellen auszuführen. Die Anweisung wird dann in die Befehle PUSH [BX+10] POP [3000] umgesetzt. Als letzte Erweiterung erlaubt der A86 die Verwendung der Segmentregister (Ausnahme CS) innerhalb eines XCHG-Befehls. Mit XCHG DS,ES generiert der Assembler die Sequenz PUSH DS, MOV DS,ES, POP ES. Der MOV-Befehl wird in obigem Fall weiter expandiert, da kein 8086-Befehl existiert.

## Lokale Labels in A86-Programmen

Eine weitere Besonderheit des A86 stellen die lokalen Labels dar. Diese Labels bestehen aus einem Buchstaben, gefolgt von einer oder mehreren Ziffern. Der A86

betrachtet diesen Namen als lokales Label, d.h., bei fehlender PUBLIC-Anweisung werden die Namen nicht als global definiert. Bei Vorwärtsreferenzen auf solche lokalen Labels muß dem Labelnamen das >-Zeichen vorangestellt werden (z.B. JMP >L1). Fehlt das Zeichen, wurde das Label bereits in einer vorhergehenden Anweisung definiert. Eric Isaacson gibt dabei an, daß ein Labelname mehrfach im Programm vorkommen kann. Ich möchte aber von einer extensiven Verwendung lokaler Labels abraten, da dies die Programmtransparenz verschlechtert.

## Diverse Optimierungen durch den A86-Assembler

Für den Profi besitzt der A86 noch einige Feinheiten. So kann bei den Befehlen AAM und AAD eine Konstante folgen, die die Basis des verwendeten Zahlensystems angibt. Normalerweise beziehen sich die Befehle auf BCD-Zahlen (Basis 10 oder 0AH). Alle Assembler kodieren die Basis für 8086-Prozessoren explizit im Code mit dem Byte 0AH. Der A86 generiert auf Wunsch die hinter der Anweisung stehende Konstante als Basis. Diese Makrofunktion sollte m.E. nicht verwendet werden, da nicht alle Prozessoren dies verkraften.

Bei der TEST-Operation sind zwei Operanden (z.B.: TEST AX,0FFFF) erforderlich. Sind im Befehl zwei gleiche Register (z.B. TEST DL,DL) angegeben, reicht dem A86 die Angabe TEST DL, um den korrekten Befehl zu generieren. Bei Memoryoperanden generiert der A86 bei fehlenden Operanden den Wert 0FFFFH. Die Anweisung TEST LEN wird dann zu TEST [LEN],0FFFF erweitert.

Bei den LEA-Instruktion läßt sich mit:

```
LEA DI, BUFFER
```

die Adresse der Variablen BUFFER in das Register DI übernehmen. Der Befehl:

```
MOV DI, OFFSET BUFFER
```

ist funktional äquivalent und zudem noch um ein Byte kürzer. Der A86 erkennt die Konstruktion und setzt automatisch den MOV-Befehl ein.

Aus meiner Sicht sind die obigen Konstruktionen im Hinblick auf die Programmtransparenz nicht sonderlich förderlich. Viele Programmierer benutzen die Befehle aber, sofern die Programme nicht MASM/TASM kompatibel sein müssen. Die Kenntnis der Optimierungen ist vor allem beim Debuggen von Programmen wichtig, weshalb ich kurz darauf eingegangen bin.

### 4.23.2 Die expliziten Makros

Bei expliziten Makros muß der Benutzer die Definition selbst anlegen. Dadurch hat er aber auch die Kontrolle über die Umsetzung und kann die einzusetzenden Befehle

selbst bestimmen. Diese Funktion wird von vielen Assemblern geboten und ist in der Regel problemlos einsetzbar.

Für die Makros gilt, daß sie vor der Benutzung definiert sein müssen. Vorwärtsreferenzen sind also nicht erlaubt. Jede Makrodefinition besteht aus dem Namen, gefolgt von dem Wort MACRO. Daran schließt sich der Text des Makros mit eventuellen Parametern an. Den Abschluß bilden die Zeichen #EM. Die Operanden eines Makros werden in der Definition mit den Parametern #1, #2 etc. markiert. Beim Aufruf des Makros muß dann die entsprechende Zahl von Parametern gesetzt werden. Das folgende Beispiel definiert ein Makro mit dem Namen CLEAR, das den Inhalt eines Registers auf 0 setzt (das Beispiel wurde gegenüber dem Original von der A86-Diskette abgewandelt, da XOR effektiver arbeitet als SUB).

```
; Makro mit einem Parameter, um ein Register zu löschen
; Format: Name Befehle #Parameter #EM
```

```
CLEAR MACRO XOR #1,#1 #EM
```

Um dieses Makro innerhalb des Programmes zu benutzen, muß nur der Name mit einem entsprechenden Parameter eingesetzt werden. Mit:

```
CLEAR AX ; generiere XOR AX,AX
CLEAR BX ; generiere XOR BX,BX
```

werden die beiden Register AX und BX auf 0 gesetzt. Der A86 kodiert dann die Makros als XOR AX,AX und XOR BX, BX. Ein weiteres Beispiel ist ein Makro zum Vertauschen von zwei Speichervariablen. Dies ist durch direkte 8086-Befehle nicht möglich.

```
CHANGE MACRO
MOV AL,#1 ; lade 1. Byte
MOV AH,#2 ; lade 2. Byte
XCHG AH,AL ; tausche
MOV #1,AL ; speichere 1. Byte
MOV #2,AH ; speichere 2. Byte
#EM
```

Sind nun zwei Byte-Variable mit:

```
LEN1 DB ?
LEN2 DB ?
```

definiert, läßt sich deren Inhalt recht einfach mit dem Makro durch:

```
CHANGE LEN1,LEN2
```

austauschen. An diesen Beispielen wird bereits die Macht der Makros deutlich. Das Format ist weitgehend frei wählbar.

Der Aufbau und die Formatierung ist recht flexibel, sofern die Randbedingungen beachtet werden. Ein Makro ist immer mit einem Namen, gefolgt von dem Schlüsselwort `MACRO` einzuleiten und mit dem String `#EM` abzuschließen. Das Makro kann in einer Zeile stehen, sollte aber bei umfangreicheren Definitionen aus Gründen der Lesbarkeit auf mehrere Zeilen aufgeteilt werden. Das Zeichen `#` (23H) besitzt innerhalb der Makrodefinition eine besondere Bedeutung, denn es leitet in der Regel die Makrobefehle (z.B. `#EM`, `#EX`, `#1`, etc.) ein. Soll nur das Zeichen `#` im Makro erhalten bleiben (z.B. in einem String), ist das Zeichen zweifach einzugeben:

```
Texte MACRO
DB '##1. Parameter' ; Text = #1.Parameter
DB '#1' ; 1. Param. übernehmen
#EM
```

Ein Aufruf des Makros mit:

```
Texte Hallo
```

erzeugt die beiden Definitionen:

```
DB '##1.Parameter'
DB 'Hallo'
```

im Quellprogramm. In der zweiten Anweisungszeile werden die Zeichen `#1` durch den 1. Parameter des Aufrufes ersetzt.

Beim Aufruf des Makros ist der Name, gefolgt von den durch Kommas getrennten Parametern anzugeben. Weiterhin dürfen Leerzeichen eingefügt werden. Um Kommata, Semikolons oder Blanks in einem Parameter zu übergeben, muß dieser in Hochkommata stehen, da der A86 sonst die Trennzeichen entfernt. Der A86 ersetzt normalerweise in der Makrozeile den angegebenen Platzhalter durch den Textstring des entsprechenden Parameters. Deshalb führt er auch keine Überprüfung auf Gültigkeit durch, die erfolgt erst bei der Übersetzung. Fehlt ein Parameter (z.B. 1. Parameter), wird der entsprechende Platzhalter entfernt (z.B. `MOV AX,#1` wird zu `MOV AX,`). Bei falschen Operatoren gibt der A86 anschließend bei der Übersetzung eine Fehlermeldung aus. Manchmal ist es jedoch erwünscht, einzelne Parameter beim Aufruf des Makros wegzulassen. Dies geschieht, indem an der betreffenden Stelle ein Komma eingetragen wird:

```
Text , 'Hallo', 123 ; 1. Parameter fehlt
Text 33,, 123 ; 2. Parameter fehlt
Text 33, 'Hallo' ; 3. Parameter fehlt
```

Das folgende Makro ermöglicht die Verwendung des Segment-Overrides in Abhängigkeit vom Parameter:

```
MOVX MACRO #1 MOV AX,[BX] #EM
```

Ein Aufruf mit MOVX CS: generiert einen Segment-Override CS:MOV AX,[BX].

Mit Makros lassen sich zum Beispiel die Befehle der 80386/80486-Prozessoren nachbilden, so daß der A86 an den erweiterten Instruktionssatz anpaßbar ist. Ein Beispiel für die Implementierung des NOP-Befehls per Makro sieht folgendermaßen aus: NOPx MACRO DB 090 #EM. Jeder Aufruf generiert einen NOP-Befehl (Code 90H).

### 4.23.3 Schleifen in Makros

Eine Erweiterung des A86-Makroprozessors erlaubt die Anwendung von Schleifen innerhalb eines Makros. Dabei wird zwischen R-Loops, die sich auf die Parameter der Aufrufzeile und C-Loops, die sich auf die einzelnen Zeichen eines Parameterstrings beziehen, unterschieden. An Stelle der Platzhalter #1 bis #9 sind die Operanden W, X, Y und Z zu benutzen.

#### Die R-LOOP

Mit dieser Variante lassen sich Parameter mehrfach verwenden. Eine R-Schleife beginnt mit dem String #R, gefolgt von dem Buchstaben des Platzhalters (W, X, Y, Z) und zwei Ziffern für den Start- und den Endeparameter. Das folgende Beispiel verdeutlicht die Anwendung:

```
SET MACRO
MOV AX,#1    ; Lade AX mit 1. Parameter
#RY24       ; Beginn R-Loop vom 2. bis 4. Parameter
MOV #Y, AX   ; Setze für Y den 2.bis 4. Parameter ein
#ER         ; Ende Loop
#EM         ; Ende Macro
```

Der Aufruf des Makros mit folgender Anweisung:

```
SET 0,BX,CX,DX
```

generiert folgende Assemblerbefehle:

```
MOV AX,0000 ; 1. Parameter
           ; Expansion durch die R-Loop
MOV BX,AX   ; 2. Parameter
MOV CX,AX   ; 3. Parameter
MOV DX,AX   ; 4. Parameter
```

Mit #RY24 wird die Variable Y sukzessive mit den Parametern 2 bis 4 geladen und dann wird der Platzhalter in den folgenden Makrozeilen durch den Inhalt von Y substituiert.

## Der L-Operator

Oft möchte man erst beim Makroaufruf die Zahl der Parameter bestimmen. Hier erlaubt es der L-Operator, die Schleife von einem Startparameter bis zum L(ast)-Parameter abzuarbeiten. Das folgende Beispiel verdeutlicht die Situation:

```
INCX MACRO
  #RZ1L ; von 1 bis LAST
  INC #Z ; Parameter
  #ER
#EM
```

Der Aufruf mit:

```
INC AX,BX,CX
```

erzeugt dann die Anweisungen:

```
INC AX
INC BX
INC CX
```

Aufrufe mit mehr oder weniger Parametern lassen sich ebenfalls absetzen. Die Schleife wird einfach bis zum letzten Parameter bearbeitet und dann abgebrochen.

## Die C-LOOP

Die zweite Variante von Schleifen läßt sich auf einzelne Zeichen eines Parameterstrings anwenden. Das Makro wird ähnlich dem R-LOOP formuliert:

```
PUSHM MACRO
  #CZ1 ; Variable Z mit 1 Zeichen
  PUSH #ZX ; ersetze Z durch Zeichen
  #EC ; Ende C-LOOP
#EM
```

Die Schleife wird mit #C begonnen und mit #EC beendet. Die Zahl 1 hinter der Variablen Z gibt an, daß 1 Zeichen aus dem Parameterstring zu lesen ist. Der Aufruf des Makros mit folgenden Parametern: PUSHM ABCD erzeugt die Sequenz:

```
PUSH AX ; 1. Zeichen = A
PUSH BX ; 2. Zeichen = B
PUSH CX ; 3. Zeichen = C
PUSH DX ; 4. Zeichen = D
```

d.h., jedes Zeichen des Parameters wird in die Variable Z abgelegt und anschließend als Makro expandiert.

## Die A-(After)- und B-(Before)-Operanden

Mit dem B-Operator läßt sich der Parameter vor dem angegebenen Operanden adressieren. Mit dem A-Operanden wird der nachfolgende Parameter verwendet. Jeder der Operanden 1..9, W, X, Y, Z kann mit A oder B kombiniert werden. Die Konstruktion #RX1BL führt die R-LOOP mit X zwischen dem 1. und dem vorletzten (BL) Parameter aus. Ähnliches gilt für A, wobei sich bei der C-LOOP die Operanden A und B auf einzelne Zeichen auswirken.

## Negative Schleifen

Negative Schleifen lassen sich in analoger Weise wie R- oder C-LOOPS mit positivem Increment konstruieren. Der Schleifenindex ist jeweils negativ zu wählen (Startwert größer Endwert). Das Ende einer negativen Schleife ist mit der Kennung #EQ abzuschließen. Ein Beispiel für die Konstruktion ist das Makro: INCM MACRO #RZL1 MOV #ZX,BX #EQ#EM, der die Parameter in umgedrehter Reihenfolge, vom letzten Parameter bis zum ersten Parameter, abarbeitet.

## Die Advance-Option

Eine weitere Optimierung besteht darin, am Schleifenende einen Wert #En für die Zahl der vorzurückenden Parameter anzugeben. Mit der Anweisung: #E2 wird am Schleifenende der Parameterzeiger um den Wert 2 verändert. Dadurch lassen sich im Makro z.B. immer 2 Parameter lesen und verarbeiten. Der kleine Makro zeigt wie dies funktioniert:

```
Daten MACRO
#RZ1L      ; R-Loop auf Z von 1 bis L
DB #X      ; n. Parameter
DB #AX     ; n+1. Parameter
#E2        ; 2 Parameter
#EM
```

Der Aufruf: Daten 1,2,3,4 erzeugt zur Assemblierungszeit die Befehle:

```
DB 1
DB 2
DB 3
DB 4
```

Bei der Anwendung auf C-LOOPS werden n Zeichen übersprungen.

## Schachtelung und Exit in Makros

Es besteht weiterhin die Möglichkeit zur Schachtelung von Schleifen innerhalb von Makros. Da nur vier Variablen W,X,Y,Z vorhanden sind, beträgt die Schach-

telungstiefe maximal 4. Soll ein Makro explizit beendet werden, läßt sich dies durch die Anweisung #EM (Exit Makro) erreichen.

## Übergabe von Werten

Der A86 expandiert einen Makro rein durch Übergabe des Textstrings. Möchte man jedoch einen Wert explizit an einen Makro übergeben und diesen Wert im Makro verarbeiten, muß die #V- (Value) Option benutzt werden. Das folgende Beispiel zeigt eine solche Anwendung:

```
JLV MACRO      ; mit V-Option
  J#1 LABEL#V2 ; V-Option für #2 !
#EM
```

Der Aufruf des Makros läßt sich dann zum Beispiel mit folgender Sequenz handhaben:

```
JINDEX = 3      ; definiere den Wert von JINDEX
JLV NC, JINDEX+1 ; erzeugt JNC LABEL4
```

Der Unterschied zur normalen Übergabe besteht darin, daß die Labelnummer aus JINDEX berechnet wird. Die übergebenen Werte müssen im Bereich zwischen 0 und 0FFFFH liegen.

## Übergabe der Operandengröße

Bei Strings als Operanden ist es manchmal erwünscht, die Länge des Strings (Operanden) zu kennen. Hierzu existiert die #Sn-Option, die die Länge in Zeichen des n. Operanden zurückgibt. Dies läßt sich zum Beispiel sehr gut benutzen, um die Länge eines Textes in einer DB-Anweisung mit abzulegen. Das Makro:

```
Text MACRO
  DB #S1, '1$'
#EM
```

löst diese Aufgabe. Der Aufruf des Makros mit der Zeile:

```
Text Hallo
```

erzeugt die Anweisung:

```
DB 5, 'Hallo'
```

d.h., der A86 ermittelt automatisch die Länge des Strings und setzt diese in die Anweisung ein.

Mit der #Nn-Option wird der angegebene Parameter n in die Textrepräsentation konvertiert (z.B. #N3 ergibt den Wert 3). In der A86 Dokumentation finden sich Beispiele für die Verwendung dieser Option.

Zudem besteht bei Schleifen (R- oder C-LOOP) das Problem, daß der Endeparameter maximal bis 9 oder bis L(ast) gehen darf. Bei Schleifen mit 100 Durchläufen muß die Zahl 100 in Klammern stehen, damit der A86 dies korrekt bearbeitet (z.B. #RY1(100) DB NY #ER). Ein Makro läßt sich jederzeit durch die #EM (Exit Makro) Anweisung unterbrechen. Eine Verwendung von Labels innerhalb eines Makros ist jederzeit möglich. Lokale Labels müssen aber an Stelle des Buchstabens L mit M (M1 bis M9) anfangen.

## 4.24 Bedingte Assemblierung

Eine weitere Funktion des A86 bildet die bedingte Assemblierung, d.h., einzelne Codebereiche lassen sich in Abhängigkeit von einer bestimmten Bedingung übersetzen. Für die bedingte Assemblierung existieren die Anweisungen:

```
#IF condition
#ELSE
#ENDIF
```

Mit der IF-Bedingung wird die bedingte Assemblierung eingeleitet, falls die Bedingung wahr ist. Ist die Bedingung falsch, läßt sich die Assemblierung im optionalen ELSEIF-Teil fortsetzen. Der Bereich der bedingten Assemblierung wird durch #ENDIF abgeschlossen. Eine Anwendung ist mit folgenden Beispiel gegeben:

```

;=====
; File: VERS.ASM (c) Born G.
; Versionsabfrage in DOS
;=====
;
        RADIX 16
        ORG 100

Vers:   JMP NEAR Start ; in Code
; Texte
Text1:
#IF PCDOS
  DB 'PC-DOS Version : $'
#ELSE
  DB 'MS-DOS Version : $'
#ENDIF
Text2:  DB 0D,0A,'$'
;
Start:  MOV AH,09          ; Textausgabe
        MOV DX,OFFSET Text1 ; an DOS
        INT 21
        MOV AH,030        ; Versionsabfrage

```

```
INT 21
MOV DL,30          ; Hauptversion in ASCII
ADD DL,AL         ; konvertieren
MOV AH,02         ; ausgeben
INT 21
MOV DL,2E         ; Punkt ausgeben
MOV AH,02         ; ausgeben
INT 21
MOV DL,30         ; Unterversion in ASCII
ADD DL,AH         ; konvertieren
MOV AH,02         ; ausgeben
INT 21
MOV AH,09         ; CRLF ausgeben
MOV DX,OFFSET Text2 ; an DOS
INT 21
MOV AX,4C00       ; EXIT
INT 21
;
END Vers
```

Listing 4.1: DOS-Versionsabfrage

Die Abfrage generiert in Abhängigkeit von der Bedingung, daß die Variable PCDOS gesetzt ist, den Text1 oder 2 und gibt die DOS-Version aus. Die Variable PCDOS muß beim Aufruf des A86 deklariert werden. Der Name der Variablen steht hinter einem Gleichheitszeichen. Der Aufruf:

```
A86 =PCDOS VERS.ASM
```

definiert die Variable PCDOS und erzeugt anschließend eine COM-Datei. Dann erscheint beim Programmaufruf der Texthinweis bezüglich der PC-DOS-Version. Wird das gleiche Programm mit der Anweisung: A86 VERS.ASM übersetzt, generiert der A86 den Text mit der MS-DOS-Meldung.

Hinter einer ELSE-Anweisung kann eine weitere IF-Anweisung geschachtelt werden (#IF Beding. #ELIF Beding. ... #ENDIF). Die bedingte Assemblierung läßt sich sehr vorteilhaft beim Debuggen von Programmen verwenden. In den Quellcode werden einfach Anweisungen zur Überprüfung des Programmablaufes (z.B. Meldungen an den Benutzer) eingesetzt und mit der bedingten Assemblierung verknüpft. Nur wenn die entsprechende Variable gesetzt wird, assembliert A86 die Testanweisungen. Die bedingte Assemblierung läßt sich auch innerhalb von Makros verwenden. Beachten Sie aber dabei, daß das Zeichen # für die Einleitung der Anweisungen (z.B.: ##IF) zu verdoppeln ist.

## 4.25 Assemblierung und Linken

Mit dem A86 lassen sich sowohl -, - als auch EXE-Dateien erzeugen. Je nach gewünschter Dateiart wird mit etwas anderen Aufrufparametern gearbeitet. Im vor-

herigen Kapitel wurde bereits kurz erläutert, wie sich Variablen zur bedingten Assemblierung in der Eingabezeile definieren lassen. Der A86 besitzt die allgemeine Form des Aufrufes:

```
A86 [=Name] Source.Ext [to] [Obj] [SYM] [LST] [XRF] [Options]
```

wobei die in eckigen Klammern angegebenen Parameter optional sind. Nach dem Gleichheitszeichen kann der Name einer internen Variablen für die bedingte Übersetzung folgen. Mit Source.Ext ist der Name der Quelldatei anzugeben. Es lassen sich durchaus mehrere Quelldateien oder Dateinamen mit Wildcards (\*.ASM, \*.8) angeben, die in einem Assemblierungslauf bearbeitet werden. Sie können zusätzlich auch die Erweiterungen für die OBJ-, Symbol- (SYM), Listdateien (LST) und Cross-Referenz-Tabellen (XRF) angeben. Im letzten Schritt können noch verschiedene Schalter für Optionseinstellungen gesetzt werden.

### 4.25.1 Erzeugung von COM-Dateien

Am einfachsten ist die Erzeugung von COM-Dateien. Der A86 ist in der Lage, lauffähige COM-Dateien in einem Schritt zu erzeugen. Bei vielen Programmen benutze ich deshalb diese Option. Um das Programm ASK.ASM aus unserem Beispiel in eine lauffähige COM-Version zu übersetzen, ist nur folgende Anweisung in DOS einzugeben:

```
A86 ASK.ASM
```

Der A86 meldet sich und gibt verschiedene Statusmeldungen aus:

```
A86 macro assembler, Vx.xx Copyright ... Eric Isaacson  
Source:  
ASK.ASM  
Object: ASK.COM  
Symbols: ASK.SYM
```

Treten bei der Assemblierung Fehler auf, integriert der A86 die Meldungen direkt in den Quellcode. Die Originaldatei wird in Name.OLD umbenannt. Ferner erscheint ein Hinweis auf die Fehler auf dem Bildschirm:

```
Error messages inserted into ask.asm  
Original source renamed as ask.old
```

Eine Liste der Fehlermeldungen findet sich im Anhang. In diesem Fall ist der Fehler im Quellprogramm zu korrigieren und die Übersetzung erneut zu starten. Bei einer fehlerfreien Assemblierung erzeugt der A86 eine COM-Datei, die sich anschließend direkt ausführen oder mit einem Debugger testen läßt. Um eine COM-Datei zu erzeugen, muß nicht unbedingt eine ORG-Anweisung im Programm vorkommen. Der

A86 generiert bei COM-Dateien den korrekten Offset von 100H. Wurde aber eine ORG-Anweisung verwendet, muß deren Wert auf 100H gesetzt werden.

### 4.25.2 Erzeugung von BIN-Dateien

BIN-Dateien werden gebraucht, wenn das Programm als DOS-Treiber fungieren soll. Die Einbindung in ältere Versionen von Hochsprachen (dBASE, Turbo Pascal 3.0, BASIC) ist teilweise nur per BIN-Datei möglich. Eine BIN-Datei wird nur erzeugt, falls der +O Schalter in der Kommandozeile fehlt und eine ORG 0 Anweisung im Programmkopf steht. Der Assembler übersetzt die Datei wie eine COM-Datei und meldet das Ergebnis (Fehlerhinweise, etc.) auf dem Bildschirm. Diese Option wird aber kaum noch verwendet.

### 4.25.3 Erzeugung von OBJ-Dateien

In vielen Fällen wird man nur Teile eines Programmes übersetzen und dann in eine sogenannte OBJ-Datei. Diese Datei läßt sich später durch den Linker mit anderen OBJ-Files zu einem lauffähigen EXE-Programm binden. Um einen OBJ-File zu erzeugen, ist entweder der Name der Zielfile explizit mit der Extension .OBJ anzugeben:

```
A86 ASK.ASM TO ASK.OBJ
```

oder der Schalter +O ist beim Aufruf zu setzen:

```
A86 ASK.ASM +O
```

In beiden Fällen legt der A86 eine OBJ-Datei mit dem Code an. Wichtig ist allerdings, daß die Datei keine ORG 100 Anweisung enthält, da der Linker die Adressen für Code und Daten festlegt.

### 4.25.4 Die A86-Optionen

Der A86 besitzt eine Reihe von Optionen zur Einstellung der Assemblierung. Nachfolgend möchte ich diese Optionen kurz vorstellen:

- +C Mit dieser Option legt der A86 alle Symbolnamen in kleinen Buchstaben in der OBJ- und SYM-Datei ab. Der Schalter emuliert den MASM /mx Schalter.
- +c Mit dem Schalter unterscheidet der Assembler zwischen Klein- und Großbuchstaben. Der Schalter emuliert den MASM /ml Switch.
- +D Mit dieser Option wird die Zahlenbasis standardmäßig auf das Dezimalsystem gestellt. Konstanten mit führenden Nullen werden aber weiterhin als Hexwerte

- erkannt.
- D Setzt die Standardeinstellung für Zahlen auf das Hexadezimalsystem, sofern führende Nullen verwendet werden. Andere Zahlen werden als Dezimalwerte interpretiert.
  - +E Erzeugt eine separate Datei Name.ERR mit den gesammelten Fehlermeldungen während der Übersetzung.
  - E Veranlaßt daß der A86 die Fehlermeldungen im Quelltext mit ausgibt. Befindet sich der zu übersetzende File nicht im aktuellen Verzeichnis, wird eine ERR-Datei generiert.
  - +F Mit dieser Option generiert der A86 die Anweisungen für den 80287-Fließkommaprozessor. Der Mode kann auch im Programm mit der Anweisung .287 spezifiziert werden.
  - +f Mit dieser Option emuliert der A86 den 8087 Befehlssatz. Sobald ein Fließkommabefehl erkannt wird, generiert der Assembler einen Aufruf für eine Fließkommabibliothek (z.B. Borland) und legt den Code in einem OBJ-File ab.
  - F Schaltet die Generierung von 8087- und 80287-Befehlen ab.
  - +L Erzwingt die Generierung von 3 Byte JMP NEAR-Befehlen an Stelle der 2 Byte JMP SHORT-Anweisungen.
  - L Erlaubt dem A86 die Generierung von 2-Byte-Sprungbefehlen.
  - +O Der A86 erzeugt mit dieser Option linkbare OBJ-Files, falls die Extension der Ausgabedatei nicht angegeben wurde.
  - O Schaltet die Erzeugung von OBJ-Files aus, wodurch COM-Files erstellt werden.
  - +S Die Option unterdrückt die Generierung der Datei mit den Symboltabellen (\*.SYM). Falls ein Name mit dem Symbolfile angegeben wird, bleibt die Option +S außer Kraft.
  - S Es wird eine Datei mit der Symboltabelle angelegt.
  - +X Weist den A86 an, daß ein undefinierter Name explizit als EXTRN zu definieren ist, andernfalls erfolgt eine Fehlermeldung. Bei COM-Files hat der Switch keine Wirkung.
  - X Mit dieser Einstellung betrachtet der A86 alle undefinierten Referenzen im

Programm als extern.

Die Schalter (Switches) lassen sich in einer Kommandozeile kombinieren +O+X, die Standardeinstellung für alle Optionen ist auf Minus gesetzt.

Als Besonderheit lassen sich die Optionen für den A86 auch über das Environment setzen. Der Umgebungsbereich wird mit dem DOS-Kommando SET verändert. Die Variable A86 kann nur Optionen, Zugriffspfade für Libraries etc. aufnehmen. Die Optionen lassen sich z.B. mit:

```
SET A86=C:\PROG\MACRO.ASM +OX
```

setzen. In diesem Fall wird immer die Option +OX eingestellt und der File mit den Makrodefinitionen MACRO.ASM übersetzt. Es dürfen durchaus mehrere Filenamen zur Übersetzung im Environmentstring angegeben werden.

Darüber hinaus können die Eingaben aus einer Datei über die DOS-I/O-Umleitung kommen. Mit der Anweisung:

```
A86 < Steuer
```

werden alle Anweisungen aus der Datei *Steuer* gelesen und nicht mehr von der Tastatur erwartet. Häufig möchte man jedoch die Dateinamen per Tastatur angeben. Findet sich in der Kommandozeile (oder im Environment) ein &-Zeichen, fragt der Assembler die Dateinamen von der Tastatur ab.

## 4.26 Linken von OBJ-Dateien

Mit der +O-Option erzeugt der A86 OBJ-Dateien, die sich mit LINK in anderen Modulen integrieren lassen. Das Programm LINK wird bei MS-DOS und verschiedenen Programmiersprachen mitgeliefert. Es kann aber auch ein anderer Linker (z.B: TLINK) verwendet werden. Das Programm besitzt die Aufrufsyntax:

```
LINK [Pfad:Files.Ext TO File.Ext Options]
```

und erlaubt drei verschiedene Aufrufvarianten. Mit LINK wird das Programm ohne Parameter aufgerufen, d.h., der Linker fragt die Dateien explizit ab:

```
Microsoft (R) Overlay Linker Version xxx  
Copyright ....
```

```
Object Modules [.OBJ]:  
Run File [.EXE]:
```

List File [.MAP]:  
Libraries [.LIB]:

In den einzelnen Eingabefeldern lassen sich dann die Modulnamen eingeben. Wird eine Eingabe mit RETURN quittiert, übernimmt LINK die Standardvorgaben.

Die zweite Aufrufform enthält alle Parameter in der Eingabezeile. Eine mögliche Anweisung ist zum Beispiel:

```
LINK ASK.OBJ TO ASK.EXE
```

um das Programm ASK.OBJ in eine lauffähige EXE-Datei zu überführen. In obigem Beispiel erscheint eine Warnung, daß der Stack fehlt. Dies kann hier ausnahmsweise ignoriert werden, in der Regel muß aber im Programmcode ein Stacksegment definiert werden.

Die dritte Aufrufform übernimmt die Linkanweisungen aus einer Datei. Hierfür muß der Dateiname mit den Anweisungen beim Aufruf angegeben werden. Zur Unterscheidung von OBJ-Modulen wird das Zeichen @ vorangestellt:

```
LINK @Make
```

In der Datei Make lassen sich nun alle Anweisungen für LINK ablegen. Weitere Hinweise zu den Link-Optionen und Fehlermeldungen finden Sie in der Microsoft Dokumentation.

Nachfolgend möchte ich aber noch an Hand eines kleinen Beispielprogramms die Prozedur von der Programmerstellung bis zum lauffähigen EXE-File zeigen.

Das Programm ESC.EXE ist ein nützliches kleines Tool, das es erlaubt, direkt eingegebene Hexzeichen und Texte an die Ausgabeeinheit zu übergeben. Damit lassen sich recht einfach Steuerzeichen zum Bildschirm oder den Drucker senden. Man denke nur an die Umstellung des Druckformates am Drucker per Batchdatei. Die folgenden Aufrufe erläutern die Möglichkeiten von ESC.EXE. Die Kommentare sind nicht Bestandteil des Aufrufes und wurden nur zur Erläuterung eingefügt.

```
ESC 07                ; erzeuge ein BELL-Signal auf dem PC
ESC 0C >PRN:         ; Seitenvorschub an Drucker
ESC 41 42,43 20 "Hallo" ; erzeugt ABC HALLO
ESC 1B 21 00 > PRN:   ; PICA Schrift am Epson LX 800
ESC "Hallo",0D 0A,"D:" ; erzeuge 2 Zeilen
```

In /2/ (siehe Literaturverzeichnis) wird die Verwendung des Tools ESC.EXE zur Erzeugung von Batchdateien zur Druckerumstellung, Menüführung, etc. detailliert besprochen.

Doch nun möchte ich kurz auf die Implementierung, die Übersetzung und den Linkvorgang eingehen. Das Programm ESC besteht aus folgenden Teilprogrammen:

```

ESC.ASM      ; Hauptprogramm mit den Prozeduraufrufen
HEXASC.ASM   ; Unterprogramm zur Wandlung ASCII-> HEX
MODULE.ASM   ; Modul mit weiteren Unterprogrammen.

```

Zuerst möchte ich kurz auf die Unterprogramme (Prozeduren) eingehen. Alle Unterprogramme sind im NEAR-Modell implementiert, d.h., der Code liegt im gleichen Segment.

Das Modul HEXASC.ASM führt die Konvertierung einer ASCII-Hexziffer in einen Hexadezimalwert aus. Das Programm besitzt folgenden Aufbau:

```

;=====
; File : HEXASC.ASM (c) G. Born
;
; Convert ASCII-> HEX
; CALL: DS:DI -> Adresse 1.Ziffer (XX)
;      CX      Länge Parameterstring
; Ret.: CY : 0 o.k.
;      AL      Ergebnis
;      DS:DI -> Adresse nächstes Zeichen
;      CY : 1 Fehler
;=====
;
;          RADIX 16          ; Hexadezimalsystem
;          PUBLIC HexAsc    ; Label global

Blank EQU 20          ; Leerzeichen
Null  EQU 30

;          CODE SEGMENT
;
HexAsc: MOV AL,[DI]    ; lies ASCII-Ziffer
        CMP AL,61     ; Ziffer a - f?
        JB L1        ; keine Kleinbuchstaben
        SUB AL,Blank  ; in Großbuchstaben
L1:     CMP AL,Null    ; Ziffer 0 - F?
        JB Error1    ; keine Ziffer -> Error
        CMP AL,46     ; Ziffer 0 - F?
        JA Error1    ; keine Ziffer -> Error
        SUB AL,Null   ; in Hexzahl wandeln
        CMP AL,9      ; Ziffer > 9?
        JBE Ok       ; JMP OK
        SUB AL,07     ; korrigiere Ziffern A..F
        JO Error1    ; keine Ziffer -> Error
Ok:     CLC          ; Clear Carry
        RET          ; Exit
; -> setze Carry
Error1: STC          ; Error-Flag
        RET          ; Exit
;
;          END HexAsc

```

*Listing 4.2: ASCII-HEX-Konvertierung*

Auf den Algorithmus möchte ich an dieser Stelle nicht eingehen, er erlaubt eine sehr effiziente Konvertierung. Die Aufrufparameter und die Ergebnisse sind im Modulkopf dokumentiert. Wichtig ist, daß am Anfang des Programmes der Name *HexAsc* als PUBLIC erklärt wird, damit der Linker die Aufrufe aus anderen Modulen auflösen kann.

Im File MODULE.ASM finden sich drei weitere Unterprogramme, die die Parameterzeile in CS:80 auf Separatoren (Blank, Komma) untersuchen, Strings separieren und ausgeben sowie die Hexzeichen einlesen, über *HexAsc* wandeln und ausgeben. Der Aufbau der Module ist den Kommentaren zu entnehmen:

```

;=====
; File : MODULE.ASM (c) G. Born
;
; File mit den Modulen zur Ausgabe und Be-
; arbeitung der Zeichen.
;=====
;
;         RADIX 16          ; Hexadezimalsystem
;         EXTRN HexAsc:NEAR ; Externes Modul
;         CODE SEGMENT
;
;=====
; SKIP Separator (Blank, Komma)
;
; Aufgabe: Suche die Separatoren Blank oder
;         Komma und überlies sie.
;
; CALL: DS:DI -> Adresse ParameterString
;         CX         Länge Parameterstring
; Ret.: CY : 0   o.k.
;         DS:DI -> Adresse 1. Zchn. Parameter
;         CY : 1  Ende Parameterliste erreicht
;=====
;
;         PUBLIC Skip
;
Skip:
Loops:AND CX,CX          ; Ende Parameterliste?
        JNZ Test1       ; Nein -> JMP Test
        STC             ; markiere Ende mit Carry
        JMP SHORT Exit2 ; JMP Exit2
Test1:  CMP BYTE [DI],20 ; Zchn. = Blank?
        JZ  Skip1       ; Ja -> Skip
        CMP BYTE [DI],2C ; Zchn. = ", " ?
        JNZ Exit1       ; Nein -> Exit1
Skip1:  DEC CX           ; Count - 1
        INC DI          ; Ptr to next Char.
        JMP NEAR Loops  ; JMP Loop
Exit1:  CLC             ; Clear Carry
Exit2:  RET
;
;-----
; Display String
;
; Aufgabe: Gib einen String innerhalb der

```

```

;           Parameterliste aus.
;
; CALL: DS:DI -> Adresse "....." String
;       CX           Länge Parameterstring
; Ret.: CY : 0   o.k.
;       DS:DI -> Adresse nach String
;       CY : 1   Ende Parameterliste erreicht
;-----
;
;           PUBLIC String
;
String:  CMP  BYTE [DI],22 ; " gefunden?
        JNZ  Exit3       ; kein String -> EXIT
        INC  DI          ; auf nächstes Zeichen
        DEC  CX          ; Count - 1
Loop1:  AND  CX,CX       ; Ende Parameterliste?
        JNZ  Test2      ; Nein -> JMP Test
        STC             ; markiere Ende mit Carry
        RET             ; Exit
Test2:  CMP  BYTE [DI],22 ; " -> Stringende?
        JZ   Ende       ; Ja -> JMP Ende
Write:  MOV  DL,[DI]    ; lies Zeichen
        MOV  AH,02      ; DOS-Code
        INT  21         ; ausgeben
        DEC  CX          ; Count - 1
        INC  DI          ; Ptr to next Char.
        JMP  NEAR Loop1 ; JMP Loop
Ende:   INC  DI          ; auf Stringende
        DEC  CX          ; Count - 1
Exit3:  CLC             ; ok-> clear Carry
        RET
;
;-----
; Display Number
;
; Aufgabe: Gib eine HEX-Zahl innerhalb der
;           Parameterliste aus.
; CALL: DS:DI -> Adresse 1.Ziffer (XX)
;       CX           Länge Parameterstring
; Ret.: CY : 0   o.k.
;       DS:DI -> Adresse nach Zahl
;       CY : 1   Ende Parameterliste erreicht
;-----
;           PUBLIC Number
;
Number: CALL NEAR HexAsc ; 1. Ziffer konvertieren
        JC  Error       ; keine Ziffer -> Error
        MOV DL,AL       ; merke Ergebnis
; 2. Ziffer lesen
        INC  DI          ; nächstes Zeichen
        DEC  CX          ; Zähler - 1
        AND  CX,CX      ; Pufferende?
        JZ  Display     ; JMP Display
        CALL NEAR HexAsc ; 2. Ziffer konvertieren
        JC  Display     ; keine Ziffer -> JMP Display
;
; schiebe 1. Ziffer in High Nibble
;

```

```

        PUSH CX          ; schiebe 1. Ziffer
        MOV CL,04       ; in High Nibble
        SHL DL,CL
        POP CX
        OR DL,AL        ; Low Nibble einblenden
Display:MOV AH,02      ; DOS-Code
        INT 21          ; Code in DL ausgeben
        AND CX,CX       ; Ende Parameterliste?
        STC            ; setze Carry zur Vorsicht
        JZ Exit4        ; Ende erreicht -> Exit
        INC DI          ; auf nächstes Zeichen
        DEC CX          ; Count - 1
        CLC            ; Clear Carry
Exit4:  RET
;-----
; Ausgabe des Fehlertextes und Exit zu DOS
;-----
Error:  MOV AH,09       ; DOS-Code
        MOV DX,OFFSET Txt ; Adr. String
        PUSH CS        ; DS = CS !!
        POP DS         ;
        INT 21         ; Ausgabe
        MOV AX,4C01    ; DOS-Code
        INT 21         ; Exit
;
;=====
; Fehlertext
;=====
Txt:    DB "Fehler in der Parameterliste",0D,0A,"$"
;
        END

```

Listing 4.3: Ausgabemodule

Hierbei ist auf Verschiedenes zu achten: einmal wird das externe Modul `HexAsc` von Unterprogrammen aufgerufen. Deshalb ist im Modulkopf der Name `HexAsc` als `EXTRN` mit dem Typ `NEAR` zu definieren. Vor jedem Unterprogramm wird der jeweilige Name als `PUBLIC` erklärt. Damit lassen sich Unterprogramme aus dem Hauptprogramm aufrufen. Die Datei wird mit der Anweisung `END` abgeschlossen. Der A86 kann durchaus mehrere Module in einer Datei übersetzen. Die `CODE SEGMENT`-Directive sorgt dafür, daß alle Anweisungen im Codesegment landen.

Das Hauptprogramm wird in der Datei `ESC.ASM` gespeichert und besitzt folgenden Aufbau:

```

;=====
; File : ESC.ASM (c) G. Born
;
; Programm zur Ausgabe von Zeichen an die
; Standardausgabeeinheit. Das Programm wird
; z.B. mit: ESC 0D,0A,1B "Hallo"
; von der DOS-Kommandoebene aufgerufen und
; gibt die spezifizierten Codes aus.
;=====

```

```

;
    RADIX 16                ; Hexadezimalsystem
    CODE SEGMENT
    Blank EQU 20           ; Blank
    EXTRN HexAsc:NEAR      ; externe Module
    EXTRN String:NEAR
    EXTRN Number:NEAR
    EXTRN Skip:NEAR

; Sprung ins Hauptprogramm
MAIN:  JMP Near Start      ; Next
;
;=====
; Definition des Stacksegmentes
;=====
;
Stack SEGMENT WORD STACK
    DW 100 DUP (?)         ; reserviere Stack
Stack ENDS
;
    CODE SEGMENT
;
;=====
; Hauptprogramm
;=====
; prüfe, ob Parameter vorhanden sind
;
Start: MOV AX,CS
      MOV DS,AX           ; DS = CS !
      CMP BYTE [80],0     ; Text vorhanden
      JZ  Endm1           ; kein Text vorhanden
      XOR CX,CX           ; clear Counter
      MOV CL,[80]         ; lade Pufferlänge
      MOV DI,0081         ; lade Puffer Offset
Loopb:                ; gebe Parameter aus
      AND CX,CX           ; Ende erreicht?
      JZ  Endm1           ; DOS-Exit
      CALL NEAR Skip      ; CALL Skip
      JC  Endm1           ; DOS-Exit
      CALL NEAR String; CALL String
      JC  Endm1           ; DOS-Exit
      CALL NEAR Skip      ; CALL Skip
      JC  Endm1           ; DOS-Exit
      CALL NEAR Number; CALL Number
      JC  Endm1           ; DOS-Exit
      JMP SHORT Loopb    ; JMP Loop
;
; DOS-Exit, Returncode in AL
;
Endm1: MOV AX,4C00        ; DOS Exitcode
      INT 21
;
      END MAIN

```

Listing 4.4: Hauptmodul ESC.ASM

Hier sind noch einige Besonderheiten zu beachten, die in den vorherigen Kapiteln bereits vorgestellt wurden. Einmal müssen im Kopf alle benutzten externen Referenzen als EXTRN erklärt werden. In diesem Fall werden nur die Namen der Unterprogramme benötigt, es könnten aber auch Variablen global vereinbart werden. Da das Programm als EXE-Datei laufen soll, muß sichergestellt werden, daß das Datensegment-Register korrekt positioniert wird. Da wir kein Datensegment benötigen, habe ich das Register auf den Wert des Codesegment-Registers initialisiert. Dadurch sind direkte Zugriffe auf den Bereich CS:80 per indirekter Adressierung möglich. Der Fehlertext muß im Codesegment liegen, da er ja initialisiert wird. Weiterhin ist ein Stacksegment explizit zu definieren. Dies erfolgt in einem eigenen Segment am Programmumfang. Mit der Directive MAIN wird das Modul als Hauptmodul erklärt, was aber nicht unbedingt notwendig ist. Nach der Erstellung der \*.ASM-Files per Texteditor sind diese mit dem A86 z.B. durch folgende Sequenz in OBJ-Files zu übersetzen:

```
A86 ESC.ASM +O
A86 MODULE +O
A86 HEXASC.ASM +O
```

Anschließend kann der Linker mit folgender Anweisung aktiviert werden:

```
LINK ESC.OBJ+MODULE.OBJ+HEXASC.OBJ TO ESC.EXE
```

Der Linker fragt dann noch einige Informationen (MAP-File, etc.) ab und erzeugt eine lauffähige EXE-Datei mit dem Namen ESC.EXE. Treten Fehler auf, erscheinen die entsprechenden Meldungen auf dem Bildschirm. Häufige Fehlerursache sind vergessene oder falsche PUBLIC- und EXTRN-Anweisungen. Alle Fehler müssen erst beseitigt werden, bevor das EXE-Programm getestet werden kann.

Um eine OBJ-Datei in einer Hochsprache einzubinden, sind die Hinweise in der jeweiligen Produktdokumentation zu beachten. Bei Turbo Pascal ab der Version 4.0 muß die OBJ-Datei mit einer entsprechenden PUBLIC-Definition vorliegen. Sie läßt sich dann mit (\$L Filename.OBJ) einbinden. Im PASCAL-Programm muß die Prozedur als EXTERNAL definiert werden (z.B. HEXASC Prozedur (Parameter) EXTERNAL;). Es ist auch auf die korrekte Parameterübergabe (per Stack) zu achten. Aus Platzgründen sei hier auf die entsprechende Dokumentation der Compilerhersteller verwiesen. In der Regel ist es aber kein Problem, mit A86 erstellte OBJ-Dateien in QuickBasic, Turbo Pascal, C, Clipper, etc. einzubinden.

**Anmerkung:** Wenn Sie das Programm ESC.EXE erstellen möchten, benötigen Sie einen Linker. Dieser ist nicht Bestandteil des A86.

## 5 Programmentwicklung mit MASM 6.x

In Kapitel 2 wurden im wesentlichen die Assembleranweisungen des 8086-Prozessors vorgestellt. Die Programme ließen sich mittels DEBUG in COM-Dateien umwandeln. Dieser Vorgang war zwar an einigen Stellen mit Mühen verbunden, hatte aber den Vorteil, daß Steueranweisungen für den Assembler nicht erforderlich waren. Im vorliegenden Kapitel möchte ich eine Einführung in die Programmentwicklung mit dem Microsoft Assembler (MASM Version 6.x) geben. Diese Einführung soll sicherlich nicht als Ersatz für die Originaldokumentation dienen. Wer täglich mit dem MASM umgeht, wird sicherlich neben den Benutzerhandbüchern auf weitere Literatur ausweichen wollen. Aber für die ersten Schritte sollten die folgenden Ausführungen eine gute Hilfe sein - was letztlich das Ziel dieses Buches ist.

Zur Steuerung des Assemblers müssen eine Reihe von Pseudobefehlen im Programm angegeben werden. Diese Befehle generieren keinen Code, sondern beeinflussen lediglich den Übersetzervorgang. Nachfolgend werde ich kurz auf diese Pseudobefehle (Directiven) eingehen. Weitere Hinweise finden sich in der Originaldokumentation des MASM.

Ein einfaches Assemblerprogramm zur Ausgabe des Textes *Hallo* besitzt dann folgenden Aufbau:

```
=====
; File:      HELLO.ASM (c) Born G.
; Version:  2.0 (ab MASM 6.0)
; Aufgabe:  Programm zur Ausgabe der
; Meldung:  "HALLO" auf dem Bildschirm.
; Das Programm ist als COM-Datei mit
; MASM zu übersetzen!!
=====
;
;          .MODEL TINY          ; COM-Datei
;          .RADIX 16           ; Hexadezimalsystem
;          .CODE
;          ORG 0100            ; Startadresse COM
;
HALLO: MOV AH,09                ; DOS-Display Code
      MOV DX,OFFSET TEXT      ; Textadresse
      INT 21                  ; DOS-Ausgabe
;
      MOV AX,4C00             ; DOS-Exit Code
      INT 21                  ; terminiere
;
;
```

```

; Textstring als Konstante
;
;
TEXT    BYTE "Hallo", 0A, 0D, "$"
;
END    Hallo

```

Listing 5.1: HELLO.ASM im MASM-Format

Ein erster Blick auf das Listing offenbart bereits, die Sache ist wesentlich komplexer als mit DEBUG. Der MASM erwartet eine Reihe von zusätzlichen Steueranweisungen, deren Sinn nicht intuitiv klar wird. Um das Programm möglichst einfach zu halten, wurde hier noch das COM-File-Format verwendet, in welches die ASM-Datei zu übersetzen ist. Dieses COM-Format stammt noch aus der CP/M-Zeit und begrenzt die Größe eines Programmes auf 64 KByte. Dies ist bei unseren kleinen Programmen sicherlich kein Handikap. Vorteile bringt allerdings die Tatsache, daß DOS beim Laden eines COM-Programms alle Segmentregister mit dem gleichen Wert initialisiert (Bild 5.1).

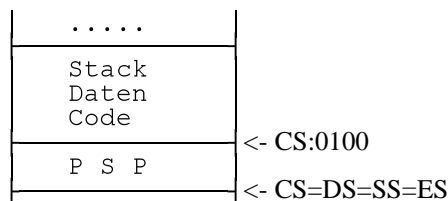


Bild 5.1: Segmentbelegung bei COM-Programmen

Ab der Adresse CS:100 beginnt der Codebereich mit den Programmanweisungen. Im gleichen Segment befinden sich auch die Daten und der Stack. Beim Zugriff auf Daten kann das Programm recht einfach aufgebaut werden. Um ein COM-Programm zu erstellen, ist obige Quelldatei mit dem Namen HELLO.ASM zu erzeugen und mit dem Befehl:

```
ML /AT /Fl HELLO.ASM
```

zu übersetzen. Der Schalter /AT sorgt dafür, daß ein COM-Programm direkt beim Linken erzeugt wird. Mit dem optionalen Schalter /Fl wird der Assembler angewiesen eine Listdatei mit dem Namen HELLO.LST anzulegen. Sobald das Programm fehlerfrei übersetzt wurde, liegt eine COM-Datei vor, die sich unter DOS ausführen läßt.

Obiges Listing enthält aber bereits eine Reihe von Steueranweisungen (Directiven), die ich nun kurz erläutern möchte.

## 5.1 Die Darstellung von Konstanten

Im Assemblerprogramm lassen sich Konstante innerhalb von Anweisungen (z.B. `MOV AX,3FFF`) eingeben. Dabei dürfen die Konstante in verschiedenen Zahlensystemen eingegeben werden. Der MASM benutzt einige implizite Konventionen:

- ◆ Standardmäßig werden Zahlen als Dezimalwerte interpretiert (z.B. 21).
- ◆ Wird an eine Zahl der Buchstabe `h` angefügt, interpretiert MASM diese Zahl als Hexadezimalwert (z.B. `3FFFh`). Eine Hexzahl muß immer mit den Ziffern 0..9 beginnen. Falls die erste Ziffer zwischen A und F liegt, ist eine 0 voranzustellen (z.B. `C000H` wird zu `0C000H`).
- ◆ Binärzahlen sind durch einen angehängten Buchstaben `y` oder `b` zu markieren. Dann sind nur die Ziffern 0 und 1 innerhalb der Zahl erlaubt.
- ◆ Dezimalzahlen lassen sich explizit durch ein angehängtes `t` (z.B: `100t`) darstellen.
- ◆ Mit dem Buchstaben `o` hinter der letzten Ziffer werden Oktalzahlen markiert.

Die Buchstaben für die Zahlenbasis können sowohl in Groß- als auch in Kleinbuchstaben angegeben werden.

## 5.2 Die RADIX-Directive

Die obigen impliziten Definitionen für die Interpretation der Zahlenbasis ist mit zusätzlichem Aufwand verbunden. Deshalb verzichten viele Programmierer auf den angehängten Buchstaben zur Markierung der Zahlenbasis. Störend ist aber die Tatsache, daß der MASM standardmäßig Zahlen zur Basis 10 interpretiert. Oft möchte man Hexadezimalzahlen eingeben. Hier läßt sich mit dem RADIX-Kommando die eingestellte Zahlenbasis überschreiben. Die Anweisung besteht aus dem Schlüsselwort und einer Zahl, die die gewünschte Zahlenbasis spezifiziert: `RADIX xx`. Die folgenden Anweisungen verdeutlichen die Anwendung des Befehls:

```
RADIX 10 ; alle Werte als Dezimalzahlen lesen  
RADIX 16 ; alle Werte als Hexadezimalzahlen  
RADIX 2  ; alle Werte als Binärzahlen lesen
```

Dem RADIX-Kommando ist ein Punkt voranzustellen. Ich habe mir angewöhnt in allen Assemblerprogrammen mit Hexadezimalzahlen zu arbeiten. Deshalb steht die RADIX-Directive auch zu Beginn eines jeden Programmes, um die Zahlenbasis 16 zu vereinbaren.

## 5.3 Definition von Datenbereichen

Häufig möchte man Variable innerhalb eines Programms anlegen. Dabei ist für jeden Eintrag genügend Speicher vorzusehen.

### 5.3.1 Datendefinitionen mit DB, DW, DD, DQ und DT

Zur Definition von Variablen kennt der MASM - wie TASM - die Pseudoanweisungen:

DB	erzeugt ein Data Byte
DW	erzeugt ein Data Word
DD	erzeugt ein Double Data Word = 4 Byte
DF	erzeugt ein FAR Data Word = 6 Byte
DQ	erzeugt ein Quad Data Word = 8 Byte
DT	erzeugt 10 Data Byte

Die DQ-Anweisung erzeugt 8 Byte, welche zur Speicherung von 8087-Werten gebraucht werden. Die DT-Anweisung erzeugt eine Variable mit 10 Byte. Vor dem Pseudobefehl kann der Name der Variablen stehen:

```
Zahlen DB 1,2,3,4,5,6  
Wort DW 0FFFFH
```

Über den Namen läßt sich später auf die Daten zurückgreifen (z.B. MOV DX,OFFSET Text). Beachten Sie, daß hinter dem Variablen kein Doppelpunkt steht.

Ab der Version 6.0 wurde im MASM jedoch eine Erweiterung der Datendefinitionen eingeführt, die einige Probleme mit älteren Programmen bringen kann. Soll ein Textstring definiert werden, konnte dies in älteren MASM-Versionen (und zumindest bei den meisten Assemblern) mit dem Befehl:

```
String DB "Dies ist ein String", 0A, 0D,"$"
```

erfolgen. Ab der Version MASM 6.0 ist Microsoft der Ansicht, daß mit DB nur ein einzelnes Byte definiert werden darf. Die obige Anweisung führt daher zu einer Fehlermeldung. Um dennoch Texte zu vereinbaren, sind die Operatoren BYTE, WORD, DWORD, QWORD, FWORD und TBYTE eingeführt worden und sollten in neuen Programmen verwendet werden.

Hier bei gilt folgende Bedeutung:

BYTE	Data Byte
WORD	Data Word

DWORD	Double Data Word = 4 Byte
FWORD	FAR Data Word = 6 Byte
QWORD	Quad Data Word = 8 Byte
TBYTE	10 Data Byte

Die Variablen werden im Assembler als vorzeichenlose Werte interpretiert. Weiterhin besteht die Möglichkeit, die Daten als Integerzahlen mit Vorzeichen zu definieren. Hierzu wird vor das betreffende Schlüsselwort der Buchstabe S gesetzt. Aus BYTE wird dann SBYTE und die Werte liegen nun zwischen -128 und +127.

Doch nun zurück zu unserem ersten MASM-Programm. Mit dem Befehl:

```
TEXT BYTE "Hallo", 0A, 0D,"$"
```

wird ein Datenbereich, bestehend aus mehreren Bytes vereinbart, der sich zusätzlich über den Namen Text ansprechen läßt. Der Assembler initialisiert den Text dann mit den angegebenen Startwerten. Mit der DB-Anweisung hätte die Variable lediglich ein Byte, aber keinen Text aufnehmen können. Beachten Sie dies bei der Erstellung von Programmen für MASM ab Version 6.0.

Konstante, die mit Werten initialisiert werden, sind im Codesegment zu speichern. Variable, deren Inhalt vom Programm verändert wird, sind dagegen im Datensegment abzulegen. Bezüglich der Definition der Segmente sei auf den Abschnitt *Die Segment Anweisung* und die folgenden Beispielprogramme verweisen.

### 5.3.2 Der DUP-Operator

Um größere Datenbereiche mit dem gleichen Wert zu initialisieren, existiert das Schlüsselwort DUP. Soll zum Beispiel ein Puffer mit 5 Byte angelegt werden, müßte die BYTE-Directive folgendes Aussehen haben:

```
Buffer BYTE 00, 00, 00, 00, 00
```

Bei diesem Beispiel läßt sich die Definition noch leicht als Quellzeile formulieren. Was ist aber, falls der Puffer 256 Zeichen umfassen soll? Es macht wohl keinen Sinn hier 256 Byte mit dem Wert 0 im Quellprogramm aufzunehmen. Die meisten Assembler bieten aber die Möglichkeit einen Datenbereich mit n Byte zu reservieren und gegebenenfalls mit Startwerten zu belegen. Um einen Puffer mit 32 Byte Länge zu vereinbaren ist folgende Anweisung möglich:

```
Buffer BYTE 20 DUP (0)
```

Beachten Sie dabei, daß die Längenangabe hier als Hexzahl (20H = 32) erfolgt. Der Puffer wird mit den Werten 00H initialisiert. Um alle Bytes auf den Wert FFH zu setzen, wäre die Anweisung:

Buffer BYTE 20 DUP (FF)

erforderlich. Das bedeutet:

- ◆ Die Zahl vor dem DUP-Operator gibt die Zahl der zu wiederholenden Elemente wieder. Beim BYTE-Operator sind dies n Byte, beim WORD-Operator n Worte.
- ◆ Nach dem DUP-Operator folgt der Initialisierungswert in Klammern.

Soll eine Variable nicht initialisiert werden, ist an Stelle des Wertes ein Fragezeichen (?) einzusetzen.

Buffer BYTE 20 DUP (?)

Der Assembler reserviert dann einen entsprechenden Speicherbereich für die Variable.

ASCII-Strings lassen sich einfach in Hochkommas einschließen und im Anschluß an eine BYTE-Anweisung eingeben:

Text BYTE 'Die ist ein Text',0D,0A

In obigem Beispiel werden Textzeichen und Hexbytes gemischt. Zur Wahrung der Kompatibilität dürfen für Textkonstante auch Hochkommas verwendet werden. Es ist aber auf die unterschiedliche Speicherung von Bytes und Worten zu achten. Die Anweisungen:

Text1 BYTE 'AB'           (speichert LOW 41H, High 42H)

Text2 WORD 'AB'           (speichert 42H, 41H)

legen die Bytes in unterschiedlichen Speicherzellen ab. Im ersten Fall steht der Wert 'A' auf dem unteren Byte, daran schließt sich der Buchstabe 'B' an. Bei der Abspeicherung von Worten ist der linke Buchstabe dem höheren Byte zugeordnet. Mit der Anweisung:

Zeiger DWORD FFFF0000

lassen sich 32-Bit-Werte (FAR Pointer) definieren.

Der DUP-Operator läßt sich übrigens auch schachteln. Die Anweisung:

Buffer BYTE 5 DUP ( 5 DUP (5 DUP (1)))

legt einen Puffer von 5 \* 5 \* 5 (also 125 Byte) an, der mit den Werten 01H gefüllt wird.

**Achtung:** Der MASM interpretiert standardmäßig alle Werte als Dezimalzahlen. Die Angaben in einer DUP-Anweisung, oder der Initialisierungswert, werden dann als Dezimalzahl ausgewertet.

Buffer 256 DUP (23)

Legt einen Puffer von 256 Byte an, der mit dem Wert 23 (dezimal) gefüllt wird. Da in der Regel bei der Assemblerprogrammierung mit Hexadezimalwerten gearbeitet wird (denken Sie nur an den INT 21 (hexadezimal)), verwende ich in meinen Programmen die RADIX-Anweisung zur Einstellung der Zahlenbasis 16. Wird dies vergessen, kommt es schnell zu Fehlern. Wer gibt schon in seinen Programmen den Befehl *INT 21H* ein. Wird die RADIX-Directive nicht verwendet, müßte der DOS-INT 21-Aufruf mit *INT 33* (dezimal) kodiert werden.

### 5.3.3 Der LENGTHOF-Operator

Häufig ist es innerhalb eines Assemblerprogramms erforderlich die Länge einer Datenstruktur zu ermitteln. Wurde ein String zum Beispiel mit:

```
Text BYTE "Hallo dies ist ein Text"
```

vereinbart, läßt sich dieser Text Byte für Byte bearbeiten. Der MASM bietet über die Directive LENGTHOF die Möglichkeit zur Abfrage der Zahl der Elemente der Variablen. Mit:

```
MOV CX, LENGTHOF Text
```

wird der Wert als Konstante dem Register CX zugewiesen. Der Befehl ist im Prinzip als *MOV CX,23T* zu interpretieren, wobei der Assembler automatisch den Wert 23T einsetzt.

Beachten Sie aber, daß nicht die Zahl der Bytes, sondern die Zahl der Elemente der Struktur mit LENGTHOF zurückgegeben wird. Bei einem WORD sind dies dann schon  $2 * 23$  Byte.

### 5.3.4 Der SIZEOF-Operator

Um die Zahl der definierten Bytes zu ermitteln, bietet der MASM die Directive SIZEOF. Wurde ein String mit:

```
Text BYTE "Hallo dies ist ein Text"
```

vereinbart, setzt die Anweisung:

```
MOV CX,SIZEOF Text
```

den Wert (Zahl der Bytes) als Konstante im Ausdruck ein. Der Befehl ist im Prinzip als `MOV CX,23T` zu interpretieren, wobei der Assembler automatisch den Wert `23T` einsetzt.

### 5.3.5 Der TYPE-Operator

Mit diesem Befehl läßt sich während der Assemblierung der Typ eines Operators abfragen. Der Befehl liefert bei einer Bytevariablen den Wert 1 und bei einem Word den Wert 2 zurück.

### 5.3.6 Die Definition von Strukturen

Mit der Anweisung `STRUC` lassen sich mehrere Variable oder Konstante zu kompakteren Datenstrukturen zusammenfassen. Die Directive erzwingt, daß die Variable zusammen abgespeichert werden und sich anschließend durch die Zeigerregister `BX`, `BP`, `DI` oder `SI` über das erste Element adressieren lassen. Die indirekte Adressierung über `[BX+SI+Konst.]` eignet sich gut zur Bearbeitung solcher Strukturen. Die Definition:

```
Buffer STRUC
  Len  Byte 80          ; Länge Puffer
  Count Byte 00        ; Zeichenzahl
  Buf  Byte 80 DUP (?) ; Bufferbereich
Buffer ENDS
```

legt eine Datenstruktur für einen Puffer an, in der das erste Byte die Pufferlänge enthält. Im zweiten Byte ist die Zahl der im Puffer befindlichen Zeichen definiert. Daran schließt sich ein Pufferbereich von 127 Byte an. Die Definition der Struktur wird mit einer `ENDS`-Directive abgeschlossen. Anwendungen für diese Technik werden in den folgenden Beispielprogrammen noch auftauchen.

## 5.4 Die MODEL-Directive

Zu Beginn des Programmes läßt sich dem Assembler mitteilen, welches Speichermodell gewählt wurde. Abgeleitet von den 8086-Speicherstruktur existierten verschiedene Optionen:

- ◆ `TINY` veranlaßt, daß Code, Daten und Stack zusammen in einem 64-KByte-Block vereint werden. Dies ist bei `COM`-Programmen erforderlich. Code und Zugriffe auf Daten werden dabei mit `NEAR`-Adresszeigern ausgeführt.

- ◆ SMALL bewirkt, daß ein 64-KByte-Codesegment und ein Datensegment gleicher Größe definiert wird. Dies setzt aber voraus, daß das Programm als EXE-File gelinkt wird.
- ◆ MEDIUM unterstützt mehrere Codesegmente und ein 64 KByte großes Datensegment. Bei dem COMPACT-Modell werden dagegen mehrere Datensegmente, aber nur ein 64-KByte-Codesegment zugelassen.
- ◆ LARGE erlaubt mehrere Daten- und mehrere Codesegmente. Hier erfolgen die Zugriffe auf Code und Daten grundsätzlich mit FAR-Zeigern.

Der MASM unterstützt weitere MODEL-Vereinbarungen, auf die ich an dieser Stelle aber nicht eingehen möchte. Unser Beispielprogramm enthält die TINY-Anweisung, damit sich das Programm in eine COM-Datei verwandeln läßt.

## 5.5 Die Segmentanweisung

Dem Assembler muß ebenfalls mitgeteilt werden, in welchem Segment er die Codes oder die Daten ablegen soll. Ein übersetztes Programm besteht minimal aus einem Code- und einem Datensegment. Bei der Erzeugung der .OBJ- und .COM-Files benötigt der MASM zusätzliche Informationen zur Generierung der Segmente (Reihenfolge, Lage, etc.). Hierzu sind mehrere Operatoren zugelassen:

- ◆ CODE oder die Anweisung CODE SEGMENT markiert den Beginn eines Programmbereiches mit ausführbaren Anweisungen. Neben den Programm-anweisungen können hier auch Konstante oder initialisierte Daten abgelegt werden. Interessant ist dies vor allem bei der Erzeugung von .OBJ-Files, da der Linker die Segmente später zusammenfaßt. Im TINY-Modell ist nur ein Codebereich erlaubt.
- ◆ DATA oder die Anweisung DATA SEGMENT signalisiert, daß ein Datenbereich folgt. Anweisungen wie DB, DW und DQ erzeugen solche Variable. Häufig findet man die Datendefinition zu Beginn eines Assemblerprogramms. Mit der ORG-Anweisung läßt sich die Lage des Datenbereiches explizit festlegen. In den Beispielprogrammen finden sich die Definitionen der Datenbereiche häufig am Programmende hinter dem Code. Dadurch legt der Assembler automatisch die Lage des Datenbereiches korrekt fest. MASM generiert dann Anweisungen, die später im Datensegment des Programmes abgelegt werden. Obiges Beispielprogramm verzichtet auf ein Datensegment, da keine Variablen zu bearbeiten sind.
- ◆ STACK oder die Anweisung STACK SEGMENT erzeugt ein Segment, welches der Linker für den Stack reserviert. In obigem Beispiel wurde auf den Stack verzichtet, da ein COM-Programm den Stack im oberen Speicherbereich des Segmentes anlegt. Mit .STACK wird automatisch 1 KByte Stack reserviert.

Weitere Beispiele für die Verwendung dieser Segmentdefinitionen finden sich in den folgenden Abschnitten.

## 5.6 Die ORG-Anweisung

Mit dieser Anweisung lassen sich Daten und Programmcode an absoluten Adressen im aktuellen Segment speichern. Bei OBJ-Dateien darf die Lage der Code- und Datenbereiche nicht festgelegt werden, da dies Aufgabe des Linkers ist. Bei COM-Programmen müssen dagegen die Adressen im Programm definiert werden. Hierzu dient die ORG-Anweisung, mit der sich die Adresse des nächsten Befehls definieren läßt. Mit `ORG 100` wird MASM angewiesen, die nächste Anweisung ab `CS:0100` zu assemblieren. Der Parameter einer ORG-Anweisung muß entweder eine Konstante oder ein Ausdruck sein. Eine Verwendung eines symbolischen Namens ist ebenfalls möglich, sofern keine Vorwärtsreferenzen dabei notwendig werden. Bei der Assemblierung einer COM-Datei generiert der MASM den Code auch ohne die ORG 100 Anweisung ab `CS:0100`. In den Beispielprogrammen wird die ORG-Anweisung jedoch verwendet. Allgemein bleibt festzuhalten, daß der ORG-Befehl nur selten im Programm auftreten sollte. Andernfalls besteht die Gefahr, daß sich Codebereiche bei späteren Programmweiterungen überlappen und damit zu Fehlern führen.

Die ORG-Anweisungen sind demnach nur bei COM-Programmen erlaubt und sinnvoll. Obiges Programm enthält z.B. eine ORG-Anweisung, so daß der Code ab `CS:100H` beginnt.

## 5.7 Der OFFSET Operator

Beim Zugriff auf Speicherzellen (z.B: `MOV AX, Buffer`) sind zwei Fälle zu unterscheiden. Einmal kann der Inhalt einer Konstanten `Buffer` gemeint sein. Andererseits besteht die Möglichkeit, die Adresse der Variablen `Buffer` in `AX` zu lesen. Um die Konstruktion eindeutig zu gestalten, existiert die Anweisung `OFFSET`, die in den Beispielprogrammen verwendet wird. Mit `MOV AX, OFFSET Buffer` wird die (Offset) Adresse der Variablen oder Konstanten *Buffer* durch den Assembler berechnet und in das Register `AX` geladen.

Die Segmentadresse läßt sich übrigens mit dem Operator:

```
MOV BX, SEG Buffer
```

ermitteln. Um den Inhalt der Variablen `Buffer` zu laden, ist die indirekte Befehlsform (z.B. `MOV AX,[Buffer]`) zu nutzen.

Programmbeispiel

Zur Auflockerung möchte ich an dieser Stelle ein weiteres Programmbeispiel in MASM einfügen. Es handelt sich um das Programm zur Vertauschung der beiden parallelen Schnittstellen LPT1 und LPT2. In MASM besitzt das Programm folgendes Format:

```

;=====
; File: SWAP.ASM (c) Born G.
; Programm zur Vertauschung von LPT1
; und LPT2. Programm als COM-Datei
; mit MASM 6.x übersetzen!!
;=====
;
; .MODEL TINY ; COM-Datei
; .RADIX 16 ; Hexadezimalsystem
; .CODE
; ORG 0100 ; Startadresse COM
;
SWAP: MOV AH,09 ; DOS-Display Code
MOV DX,OFFSET TEXT ; Textadresse
INT 21 ; DOS-Ausgabe
;
MOV AX,0000 ; ES auf Segm. 0000
MOV ES,AX ; setzen
MOV AX,ES:[0408] ; Portadresse LPT1
MOV BX,ES:[040A] ; Portadresse LPT2
XCHG AX,BX ; Swap Adressen
MOV ES:[0408],AX ; store Portadressen
MOV ES:[040A],BX ;
;
MOV AX,4C00 ; DOS-Exit Code
INT 21 ; terminiere
;
; .DATA
;
; definiere Textstring im Datenbereich
;
TEXT BYTE 'SWAP LPT1 <-> LPT2 (c) Born G.',
0D,0A,'$'
;
END Swap

```

Listing 5.2: SWAP.ASM

Das Programm wird wieder als COM-Datei übersetzt. Als Erweiterung zur Version aus Kapitel 2 gibt das Programm beim Aufruf noch eine Meldung an den Benutzer aus. Die Textkonstante wird in diesem Beispiel explizit im Datensegment abgelegt. Die Anweisung `.DATA` definiert den Beginn des Segmentes. Auch hier ist wieder eine Besonderheit bei Verwendung des Segment-Override-Operators zu beachten.

## 5.8 Die Segment-Override-Anweisung

In Kapitel 2 wurde die Adressierung der 8086-Befehle besprochen. Standardmäßig greift der Prozessor auf das Datensegment zu. Bei Adressierung über das BP-Register finden sich die Daten im Stacksegment. Mit der Segment-Override-Anweisung läßt sich die Standardeinstellung für die Segmentierung für den folgenden Befehl aussetzen. In DEBUG muß der Segmentdescriptor vor dem Befehl stehen:

```
ES:MOV AX,[0408]    ; Portadresse LPT1
ES:MOV BX,[040A]    ; Portadresse LPT2
```

Der MASM generiert bei diesem Konstrukt eine Fehlermeldung, erwartet er doch den Segmentdescriptor vor der eckigen Klammer.

```
MOV AX,ES:[0408]    ; Portadresse LPT1
MOV BX,ES:[040A]    ; Portadresse LPT2
```

Sollte der Assembler einen Fehler in einer Anweisung mit Segment-Override melden, prüfen Sie bitte zuerst, ob obige Konventionen erfüllt sind.

### Programmbeispiel

Als nächstes Beispiel möchte ich das Programm zur Abschaltung der *NumLock*-Taste in der Version für den MASM vorstellen. Das Programm besitzt folgenden Aufbau:

```

;=====
; File: NUMOFF.ASM   (c) Born G.
; Ausschalten der NUM-Lock-Taste.
; Programm als COM-Datei mit MASM 6.x
; übersetzen!!
;=====
;
;      .MODEL TINY          ; COM-Datei
;      .RADIX 16           ; Hexadezimalsystem
;
;      Programmkonstanten definieren
;
;      SEG0    EQU 0000    ; 1. Segment
;      Key     EQU 0DF     ; NumLock Key
;      Key_Adr EQU 0417    ; Adresse BIOS
;      Exit    EQU 4C00    ; DOS-Exit
;      DOS_Txt EQU 09      ; DOS-Text
;
;      .CODE
;      ORG 0100           ; Startadresse COM
;
NUMOFF:MOV AH,DOS_Txt     ; DOS-Display Code
        MOV DX,OFFSET TEXT ; Textadresse
        INT 21            ; DOS-Ausgabe
;

```

```

        MOV AX,SEG0          ; ES auf BIOS-Segm.
        MOV ES,AX           ; setzen
; Bit der NumLock-Taste ausblenden
        AND BYTE PTR ES:[Key_Adr],Key
;
        MOV AX,Exit         ; DOS-Exit Code
        INT 21              ; terminiere
;
        .DATA
;
; definiere Textstring im Datenbereich
TEXT   BYTE 'NUMOFF (c) Born G.',0D,0A,'$'
;
END NUMOFF

```

*Listing 5.3: NUMOFF.ASM*

Auch hier ergibt sich wieder ein neuer Aspekt. Innerhalb des Programmes werden verschiedene Konstanten verwendet (z.B: MOV AH,09 für die Textausgabe). Ändert sich der Wert einer Konstanten, ist gerade bei größeren Programmen die Modifikation sehr fehlerträchtig. Alle Anweisungen, in denen die Konstante vorkommt, müssen gesucht und modifiziert werden. Mit der EQU-Directive läßt sich hier eine bessere Lösung finden.

## 5.9 Die EQU Directive

Mit EQU läßt sich einem Namen eine Konstante, ein Ausdruck, etc. zuweisen. Immer wenn im Verlauf des Programmes dann dieser Name auftaucht, ersetzt der Assembler diesen Namen durch den mit EQU zugewiesenen Wert. Statt der Anweisung: MOV AX,0FFFF läßt sich das Programm wesentlich transparenter mit der Sequenz:

```

True EQU 0FFFF           ; Konstante True
False EQU 0              ; Konstante False
.
MOV AX,True              ; AX Init

```

gestalten. Ähnlich lassen sich Masken, Pufferlängen, etc. mit symbolischen Namen belegen und per EQU definieren. Im Programm taucht dann nur noch der symbolische Name auf. Werden die EQU-Anweisungen im Programmkopf definiert, lassen sich die Programme sehr transparent gestalten, da sich diese EQU's leicht ändern lassen.

In obigem Programmbeispiel wurden alle Konstanten mit EQU am Programmanfang definiert. In der Praxis definiere ich nur die wichtigsten Konstanten im Programmkopf mit EQU. Bei der Anweisung:

```

MOV AH,09
INT 21

```

ist für mich sofort ersichtlich, daß es sich hier um eine Textausgabe unter DOS handelt. Die im Beispiel verwendete Sequenz:

```
NUMOFF:    MOV AH,DOS_Txt    ; DOS-Display Code
           MOV DX,OFFSET TEXT ; Textadresse
           INT 21           ; DOS-Ausgabe
```

ist nach meiner Ansicht nicht so transparent. Letztlich ist es aber Geschmacksache, welche Konstante direkt im Programm eingesetzt werden und welche global im Programmkopf definiert werden.

Zusätzlich lassen sich auch Ausdrücke mit EQU definieren. Die Anweisungen:

```
Len EQU 10T
Step EQU 3
Lang EQU Len * Step
```

weist der Konstanten Lang den Wert 30 (dezimal) zu.

Weiterhin lassen sich die EQU-Definition verwenden um einen Interrupt mit einem symbolischen Namen zu versehen. Der INT 3 wird häufig von Debuggern zum Test verwendet. Die CPU führt im Single-Step-Mode nach jedem Befehl einen INT 3 aus. Es besteht daher die Möglichkeit, den INT 3 durch:

```
TRAP EQU 3
```

umzudefinieren. Immer wenn im Programm der Begriff INT TRAP auftritt, ersetzt der Assembler diesen durch die INT 3-Anweisung.

Mit EQU-Anweisungen kann man in den meisten Assemblern einem Namen durchaus mehrfach verschiedene Werte innerhalb des Programmes zuweisen. Im MASM sind Mehrfachdeklarationen mit EQU unzulässig. Mit dem Operator = kann diese Restriktion allerdings umgangen werden. Er ist als Synonym für EQU zu verwenden. Dann können einem Namen im Verlauf des Programms durchaus unterschiedliche Werte zugewiesen werden.

## 5.10 Operationen auf Ausdrücken

Bei der Erstellung von Assemblerprogrammen werden häufig Konstante in Ausdrücken verwendet (z.B: AND AX,3FFF). Die Konstante ist dabei in der geeigneten Form im Programm anzugeben. Eine Möglichkeit besteht darin, als Programmierer den Wert der Konstanten zu berechnen und im Quellprogramm einzusetzen. Dies ist aber nicht immer erwünscht: so kann es durchaus fehlerträchtig

sein, wenn mehrere Werte manuell addiert werden. Um das Programm möglichst transparent zu gestalten, möchte man häufig die ursprünglichen Teilwerte im Programm mit angeben. Die meisten Assembler unterstützen diese Form und berechnen zur Übersetzungszeit den Wert eines Ausdrucks. Der MASM bietet eine Reihe solcher Operatoren, die nachfolgend kurz aufgeführt werden.

### 5.10.1 Addition

Der Operator erlaubt die Addition mehrerer Konstanten innerhalb eines Ausdrucks. Die Addition kann dabei durch ein + Zeichen markiert werden.

```
K22 EQU 20 + 02
Muster EQU Muster1 + Muster2
MOV AX, 04C00 + 22 ; Berechne Konstante
```

Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen, sowie Kommazahlen verwendet werden.

### 5.10.2 Subtraktion

Der Operator erlaubt die Subtraktion von Konstanten innerhalb eines Ausdrucks. Die Subtraktion wird dabei durch das - Zeichen markiert (z.B. MOV AX, 033-030). Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen, sowie Kommazahlen verwendet werden. Bei Variablen muß der Typ der beiden Operatoren übereinstimmen.

### 5.10.3 Multiplikation und Division

Die Operatoren erlauben die Multiplikation und Division von Konstanten innerhalb eines Ausdrucks. Die Operatoren dürfen nur mit Kommazahlen oder auf ganzen Zahlen durchgeführt werden.

```
CMP CL, 2 * 3 ; Compare mit 6
MOV DX, 256 / 16 ; lade DX mit 16
MOV BX, 4 MOD 2 ; lade BX mit 0
```

Mit Hilfe dieser Operatoren lassen sich Berechnungen zur Assemblierungszeit ausführen. Dies erspart die manuelle (und fehleranfällig) Berechnung durch den Programmierer. Weiterhin wird im Listing sofort sichtbar wie die Konstante berechnet wird.

### 5.10.4 Schiebeoperatoren (SHL, SHR)

Mit den Operatoren SHR und SHL lassen sich Schiebeoperationen auf einer Konstanten oder einem Ausdruck ausführen. Die Operatoren besitzen das Format:

Operand SHR Count (shift right)  
Operand SHL Count (shift left)

Die Schiebefehle erlauben es, den Operanden bitweise nach links oder rechts zu verschieben. Der zweite Operand *Count* gibt dabei die Zahl der zu verschiebenden Binärstellen an. Die Bits, die in den Operanden eingeschoben werden, sind mit dem Wert 0 belegt. Beispiel für die Anwendung der Operatoren sind:

```
MOV BX,0FF33 SHR 4; BX = 0FF3  
MOV DX,01 SHL 4 ; DX = 010
```

### 5.10.5 Logische Operatoren

Mit den Anweisungen AND, OR, XOR und NOT lassen sich logische Operationen auf den Operanden ausführen. Die Operatoren besitzen das Format:

Operand AND Operand  
Operand OR Operand  
Operand XOR Operand  
NOT Operand

Die Befehle dürfen ausschließlich mit vorzeichenlosen Byte- oder Word-Konstanten benutzt werden. Die Anweisung: MOV AL,03F AND 0F blendet die oberen 4 Bits der Konstanten 3FH aus. Für die Verknüpfung der Operanden gelten die Regeln für AND, OR, XOR und NOT. Der NOT-Operator invertiert den Wert der angegebenen Konstanten.

### 5.10.6 Vergleichsoperatoren

Der MASM bietet einen weiteren Set an Vergleichsoperatoren:

Operand EQ Operand (equal)  
Operand NE Operand (not equal)  
Operand LT Operand (less than)  
Operand LE Operand (less or equal)  
Operand GT Operand (greater then)  
Operand GE Operand (greater then or equal)

Als Operanden müssen vorzeichenlose Ganzzahlen (Byte oder Word) angegeben werden. Dabei können sowohl Konstante als auch Variable benutzt werden (z.B. MASKE1 EQ Mode). Als Ergebnis wird ein Byte oder Word zurückgegeben, welches die Werte true (0FFFFH) oder false (0) enthält. Die Anweisung: MOV AL, 4 EQ 3 lädt AL mit dem Wert 0, da der Ausdruck falsch ist.

Die bisher besprochenen Operanden lassen sich in Ausdrücken kombinieren. Dabei gelten die folgenden Prioritäten:

1. Klammern
2. Punkt
3. Segment Override, PTR
4. OFFSET, SEG, TYPE
4. HIGH, LOW
5. +, -
6. \*, /, MOD, SHR, SHL
7. +, - (binär)
8. EQ, NE, LT, LE, GT, GE
9. NOT
10. AND
11. OR, XOR
12. SHORT

bei der Auswertung eines Ausdruckes. Die Zahl 1 definiert dabei die höchste Priorität.

## 5.11 Die EVEN Directive

Mit dieser Anweisung erzwingt der Programmierer, daß der nächste assemblierte Befehl oder die Lage einer Variablen auf einer geraden Speicheradresse festgelegt wird. Innerhalb des Codesegmentes fügt der MASM gegebenenfalls eine NOP-Anweisung ein. Bei Variablen werden diese einfach auf eine gerade Adresse gelegt. Die EVEN-Directive ist besonders bei 80X86-Prozessoren hilfreich, da Zugriffe auf gerade Adressen schneller ausgeführt werden. Ein 16-Bit-Zugriff auf eine ungerade Adresse erfordert 2 Speicheroperationen.

### Programmbeispiel

Doch nun möchte ich ein weiteres Programmbeispiel vorstellen. Erstmals soll nun eine EXE-Datei erstellt werden. Das Programm besitzt den Namen:

ASK.EXE

und erlaubt Benutzerabfragen aus Batchdateien. Das Programm wird mit:

ASK <Text>

aufgerufen. Der Text innerhalb der Aufrufzeile ist dabei optional. Nach dem Start wird der Text der Aufrufzeile am Bildschirm ausgegeben. Dann wird die Tastatur abgefragt und das Zeichen der gedrückten Taste wird an DOS zurückgegeben. Der Tastencode läßt sich innerhalb der Batchdatei dann durch die ERRORLEVEL-Funktion abfragen.

Mit dem Aufruf:

ASK Abbruch (J/N) ?

gibt ASK die Nachricht:

Abbruch (J/N)?

aus und wartet auf eine Benutzereingabe. ASK terminiert sofort nach der Eingabe. Der Code des Zeichens läßt sich in DOS dann per ERRORLEVEL abfragen:

```
ASK Abbruch (J/N)?
IF ERRORLEVEL 75 GOTO NO
IF ERRORLEVEL 74 GOTO YES
:NO .....
```

Zu beachten ist lediglich, daß ERRORLEVEL die Codes zwischen 0 und 255 auf >= abprüft. Beispiele für den Einsatz des Programmes ASK finden sich im Anhang und in /2/. Doch nun möchte ich das dazugehörige Listing vorstellen.

```
=====
; File: ASK.ASM    (c) Born G.
; Version: V 1.0 MASM 6.x
; Funktion: Programm zur Benutzerab-
; frage in Batchdateien. Aufruf:
;
;     ASK <Text>
;
; Der Text wird auf dem Screen ausge-
; geben. Der Tastencode wird an DOS
; zurückgegeben. (Bei Funktionstasten
; wird FF zurückgegeben). Er läßt sich
; per ERRORLEVEL abfragen. Das Programm
; ist als EXE-Datei zu übersetzen!!
=====
;
;     .MODEL SMALL
;     .RADIX 16                ; Hexadezimalsystem
;
;     Blank EQU 20             ; Blank
;     Err1  EQU 0FF            ; Error
;
;     .STACK                   ; 1 K Stack
```

```

        .DATA
;
; Bereich mit den Textkonstanten
;
Crlf  BYTE 0D, 0A,"$"
;

        .CODE
;
ASK:   CALL NEAR PTR Text    ; Textausgabe
        CALL NEAR PTR Key    ; Abfrage der Tastatur
        PUSH AX              ; merke Tastencode
;
; CR,LF ausgeben
;
        MOV AX,SEG Crlf     ; lese Segment Text
        PUSH DS             ; merke DS-Inhalt
        MOV DS,AX          ; setze Segmentadr.
        MOV AH,09          ; INT 21-Stringausgabe
        MOV DX,OFFSET Crlf ; Stringadresse
        INT 21             ; ausgeben
        POP DS             ; restauriere DS
        POP AX             ; restauriere Tastencode
;
; DOS-Exit, Returncode steht bereits in AL
;
        MOV AH,4C          ; INT 21-Exitcode
        INT 21            ; terminiere
;

Text   PROC NEAR
;-----
; Unterprogramm zur Ausgabe des
; Textes aus der Kommandozeile
;-----
; ermittle Lage des PSP über undok.
; Funktion 51 des INT 21
;
        MOV AH,51          ; DOS-Code
        INT 21            ; Get PSP
        MOV ES,BX         ; ES:= PSP-Adr
;
; prüfe ob Text im PSP vorhanden ist
;
        MOV CL,ES:[80]    ; lese Pufferlänge
        CMP CL,0          ; Text vorhanden ?
        JZ  Ready         ; Nein -> Exit
;
; Text ist vorhanden, ausgeben per INT 21, AH = 02
;
        MOV BX,0082       ; Zeiger auf 2. Zeichen
        DEC CL            ; 1 Zeichen weniger
Loop1: ; Beginn der Ausgabeschleife !!!!
        MOV AH,02         ; INT 21-Code Display Char.
        MOV DL,ES:[BX]    ; Zeichen in DL laden
        INT 21           ; CALL DOS-Ausgabe
        INC BX           ; Zeiger nächstes Zchn
        DEC CL           ; Zeichenzahl - 1
        JNZ Loop1        ; Ende ? Nein-> Loop

```

```

;
    MOV AH,02          ; INT 21-Code Display
    MOV DL,Blank      ; Blank anhängen
    INT 21            ; und ausgeben
Ready: RET           ; Ende Unterprogramm
Text  ENDP

;
Key   PROC NEAR
;-----
; Unterprogramm zur Tastaturabfrage
; benutze INT 21, AH = 08 Read Keyboard
; oder:      AH = 01 Read Keyboard & Echo
;-----
; lese 1. Zeichen
    MOV AH,01          ; INT 21-Read Key & Echo
    INT 21            ; Read Code
    CMP AL,0          ; Extended ASCII-Code ?
    JNZ Exit         ; Nein -> Ready
;
; lese 2. Zeichen beim Extended ASCII-Code
;
    MOV AH,08          ; INT 21 Read Keyboard
    INT 21            ; Code aus Puffer lesen
    MOV AL,Err1       ; Fehlercode setzen
Exit:  RET           ; Ende Unterprogramm
Key   ENDP

END Ask

```

*Listing 5.4: ASK.ASM*

Das Programm benutzt eine Reihe neuer Funktionen, auf die ich kurz eingehen möchte. Zuerst einmal die Technologie der DOS-Aufrufe zur Erledigung der Aufgabe. Zur Ausgabe der Benutzermeldung ist die INT 21-Funktion AH = 02 zu nutzen, die ein Zeichen im Register DL erwartet und dieses Zeichen ausgibt. Dann fragt ASK die Tastatur ab. Mit der INT 21-Funktion AH = 01H wartet DOS solange, bis ein Zeichen eingegeben wird. Das Zeichen wird bei der Eingabe durch DOS auf den Ausgabebildschirm kopiert. So sieht der Benutzer seine Eingabe. Nun besitzt ein DOS-Rechner Tasten, die nicht ein Byte, sondern zwei Byte (Extended ASCII Code) zurückgeben. Funktionstasten gehören zu dieser Kategorie. Wird eine Taste mit erweitertem Code (Extended ASCII-Code) betätigt, hat das erste gelesene Byte den Wert 00H. Dann muß das Programm den Tastaturpuffer ein weiteres mal lesen um das zweite Zeichen zu entfernen. Hierzu wird die INT 21-Funktion 08H benutzt, die kein Echo der eingegebenen Zeichen zuläßt. Um innerhalb eines Batchprogramms zu erkennen, daß eine Funktionstaste gedrückt wurde, sollte ASK in diesem Fall den Wert FFH an DOS zurückgeben. Damit entfallen die Funktionstasten zur Eingabe. Die INT 21-Funktionen AH = 01H und AH = 08H geben das gelesene Zeichen im Register AL zurück. Wird nun die INT 21-Funktion AH = 4CH aufgerufen, terminiert das Programm ASK. Der Code im Register AL läßt sich dann über ERRORLEVEL abfragen. Mit diesem Wissen sollte sich das Problem eigentlich lösen lassen.

Leider bringt die Verwendung von EXE-Dateien eine Reihe weiterer Schwierigkeiten mit sich. So kann die Lage des PSP nicht mehr so einfach wie bei COM-Dateien ermittelt werden. Vielmehr ist der (undokumentierte) INT 21-Aufruf AH=51H zu nutzen. Dieser gibt im Register BX die Lage des PSP-Segementes zurück.

## 5.12 Die PROC Directive

Um das Programm ASK transparenter zu gestalten wurden einzelne Funktionen in Unterprogramme verlagert. Die Ausgabe des Textes der Kommandozeile erfolgt in dem Unterprogramm *Text*. Für die Tastaturabfrage ist das Modul *Key* zuständig. Die Prozeduren werden mit den Schlüsselworten:

```
xxx PROC NEAR
....
xxxx ENDP
```

eingeschlossen. Damit erkennt der Assembler, daß er ein RET-NEAR als Befehl einsetzen muß. Beim Aufruf der Prozedur kann MASM auch prüfen, ob der Abstand größer als 64 KByte wird.

Diese Anweisung muß bei MASM am Beginn eines Unterprogrammes (Prozedur) stehen. Dabei sind die Schlüsselworte:

```
Name PROC FAR
Name PROC NEAR
Name PROC
```

zulässig. *Name* steht dabei für den Namen der Prozedur, während die Schlüsselworte FAR und NEAR die Aufrufform für den CALL-Befehl festlegen. Sobald ein RET-Befehl auftritt, ersetzt der Assembler diesen durch die betreffende RET- oder RETF-Anweisung. Der Programmierer braucht also nur einen Befehlstyp für RET einzusetzen und der Assembler ergänzt den Befehl. Viele Programmierer ziehen aber vor, den RET-Befehl explizit (notfalls auch als RETF) zu schreiben. Dies ist bei der Wartung älterer Programme hilfreich, da sofort erkennbar wird, ob eine Prozedur mit FAR oder NEAR aufzurufen ist. Wird eine Prozedur mit PROC definiert, muß am Ende des Programmes das Schlüsselwort ENDP stehen. Da das SMALL-Model vereinbart wurde, sind alle Unterprogrammaufrufe als NEAR kodiert.

**Anmerkungen:** Bei einem EXE-Programm läßt sich ein Stack mit der STACK-Directive erzeugen. Ein Problem ist die Zuweisung eines korrekten Wertes für die Segmentregister (DS, SS). Der MASM bietet die beiden Directiven:

```
.STARTUP
...
```

.EXIT

die die erforderlichen Anweisungen erzeugen. In obigem Programmbeispiel wurde allerdings darauf verzichtet. Vielmehr wurden die Segmentregister manuell gesetzt.

Um die EXE-Datei zu erzeugen, sind mehrere Schritte erforderlich:

- ◆ Erstellen der Quelldatei
- ◆ Übersetzen in einen OBJ-File
- ◆ Linken des OBJ-Files zu einer EXE-Datei

Die Übersetzung der Quelldatei in eine OBJ-Datei erfolgt mit der Anweisung:

```
ML /FI ASK.ASM
```

Der Assembler legt seine Fehlermeldungen dann in der Listdatei ASK.LST ab. Bei einer fehlerfreien Übersetzung ist die OBJ-Datei in eine EXE-Datei zu linken:

```
LINK ASK.OBJ
```

Der Linker wird dann die Namen der EXE-, MAP- und LIB-Dateien abfragen. Um diese Abfrage zu umgehen, kann hinter dem OBJ-Namen eine Serie von Kommas folgen:

```
LINK ASK.OBJ,,,
```

Damit wird LINK signalisiert, daß die Namen der jeweiligen Ausgabedateien aus dem Namen der Eingabedatei (hier ASK) zu extrahieren ist.

Sobald die Datei ASK.EXE vorliegt, läßt sie sich mit einem Debugger testen.

### 5.12.1 Erstellen einer EXE-Datei aus mehren OBJ-Files

Nun möchte ich noch einen Schritt weitergehen und ein EXE-Programm aus mehreren Modulen aufbauen. Bei größeren Programmen möchte man nicht alle Unterprogramme in einer Quelldatei stehen haben. Vielmehr wird man einzelne Unterprogramme und Module in getrennten Dateien halten. Dies erleichtert das Editieren und Übersetzen. Sobald die Datei in einen OBJ-File übersetzt wurde, läßt sich dieser mit LINK zu den Hauptprogrammen zubinden. Diese Technik möchte ich kurz vorstellen.

#### Programmbeispiel

Unser Beispielprogramm erhält den Namen:

## ESC.EXE

und erlaubt die Ausgabe von Zeichen an die Standard-Ausgabeeinheit. Die Zeichen lassen sich als Strings oder Hexzahlen beim Aufruf eingeben. Damit gilt folgende Aufrufsyntax:

```
ESC <Param1> <Param2> .... <Param n>
```

Die Parameter <Param x> enthalten die auszugebenden Zeichen als:

- ◆ Hexzahl
- ◆ String

Die Parameter sind durch Kommas oder Leerzeichen zu trennen. Ein String zeichnet sich dadurch aus, daß die Zeichen durch Anführungszeichen eingerahmt werden. Mit ESC lassen sich zum Beispiel sehr komfortabel Steuerzeichen an den Drucker übergeben. Folgende Zeilen enthalten Beispiele für den Aufruf des Programmes. Die Kommentare sind nicht Bestandteil des Aufrufes:

```
ESC 41 42 43           ; erzeuge ABC auf Screen
ESC 0C > PRN:         ; Seitenvorschub auf PRN:
ESC 1B 24 30 > PRN:   ; Steuerzeichen an PRN:
ESC "Hallo" 0A 0D     ; Hallo auf Screen
ESC "Text" > A.BAT    ; in Datei schreiben
```

In der Vergangenheit hat mir das Programm gute Dienste geleistet. So läßt sich mit einigen Batchbefehlen der Drucker auf beliebige Schriftarten einstellen. Im Anhang ist ein Batchprogramm als Beispiel enthalten. Weitere Hinweise finden sich in /2/.

Das Programm ESC.EXE wird in drei Quelldateien aufgeteilt. Eine Datei enthält ein Unterprogramm zur Wandlung einer Hexzahl in Form eines ASCII-Strings in den entsprechenden Hexadezimalwert. Die Quelldatei besitzt folgendes Format:

```

;=====
; File : HEXASC.ASM (c) G. Born
; Version: 1.0 (MASM 6.x)
; Convert ASCII-> HEX
; CALL: ES:DI -> Adresse 1.Ziffer (XX)
;      CX      Länge Parameterstring
; Ret.: CY : 0 o.k.
;      AL      Ergebnis
;      ES:DI -> Adresse nächstes Zeichen
;      CY : 1 Fehler
;=====
;
      .MODEL SMALL
      .RADIX 16           ; Hexadezimalsystem
      PUBLIC HexAsc     ; Label global

```

```

Blank EQU 20           ; Leerzeichen
Null EQU 30

        .CODE
;
HexAsc: MOV AL,ES:[DI] ; lese ASCII-Ziffer
        CMP AL,'a'     ; Ziffer a - f ?
        JB L1         ; keine Kleinbuchstaben
        SUB AL,Blank   ; in Großbuchstaben
L1:     CMP AL,Null     ; Ziffer 0 - F ?
        JB Error1     ; keine Ziffer -> Error
        CMP AL,'F'    ; Ziffer 0 - F ?
        JA Error1     ; keine Ziffer -> Error
        SUB AL,Null   ; in Hexzahl wandeln
        CMP AL,9      ; Ziffer > 9 ?
        JBE Ok        ; JMP OK
        SUB AL,07     ; korr. Ziffern A..F
        JO Error1     ; keine Ziffer -> Error
Ok:     CLC           ; Clear Carry
        RET           ; Exit
; -> setze Carry
Error1: STC           ; Error-Flag
        RET           ; Exit
;
        END HexAsc

```

*Listing 5.5: HEXASC.ASM*

Das Unterprogramm erwartet in ES:DI einen Zeiger auf das erste Zeichen der zu konvertierenden Zahl. HEXASC konvertiert immer zwei Ziffern zu einem Byte. Das Ergebnis wird in AL (als Byte) zurückgegeben. Ist das Carry-Flag gelöscht, dann ist der Wert gültig. Bei gesetztem Carry-Flag wurde eine ungültige Ziffer gefunden und das Ergebnis in AL ist undefiniert.

In einer zweiten Quelldatei wurden zusätzliche Hilfsmodule untergebracht:

## SKIP

Aufgabe dieses Moduls ist es, innerhalb des Eingabestrings die Separatorzeichen Blank und Komma zu erkennen. Der Eingabestring wird durch ES:DI adressiert. Erkennt SKIP einen Separator, wird der Lesezeiger solange erhöht, bis kein Separatorzeichen mehr vorliegt oder das Ende des Strings erreicht wurde. Wurde das Ende des Strings erreicht, markiert SKIP dies durch ein gesetztes Carry-Flag. Der Lesezeiger ES:DI zeigt nach dem Aufruf immer auf das nächste Zeichen.

## String

Kommt in der Eingabezeile Text vor:

ESC "Hallo" 0D 0A

wird *String* benutzt um den Text direkt an die Ausgabeinheit auszugeben. Die Funktion erwartet den Lesezeiger auf den Text in ES:DI. Wird im ersten Zeichen ein " gefunden, gibt das Unterprogramm die folgenden Zeichen aus. Das Unterprogramm terminiert, sobald ein zweites Anführungszeichen auftritt, ohne daß das String-Ende erreicht ist. Wird das Ende der Parameterliste erreicht, ohne daß der String beendet wurde (zweites Anführungszeichen " fehlt), gibt das Modul ein gesetztes Carry-Flag zurück. Nach der Ausgabe des Strings zeigt der Lesezeiger auf das nächste Zeichen hinter dem String.

## Number

Dieses Modul wertet die Parameterzeile aus und liest eine Hexadezimalzahl mit 2 Ziffern ein. Anschließend wird das Unterprogramm HEXASC zur Konvertierung aufgerufen. Der zurückgegebene Wert wird dann an die Ausgabeinheit gesendet. Das Unterprogramm erwartet wieder einen Lesezeiger in den Registern ES:DI. Nach dem Aufruf wird das Zeichen hinter der Hexzahl durch diesen Zeiger adressiert. Ist das Carry-Flag gesetzt, wurde das Ende der Parameterzeile erreicht. Falls ein fehlerhaftes Zeichen erkannt wird, terminiert das Programm über die DOS-Exit-Funktion.

Das Quellprogramm besitzt folgenden Aufbau:

```

;=====
; File : MODULE.ASM (c) G. Born
; Version: 1.0 (MASM 6.x)
; File mit den Modulen zur Ausgabe und Be-
; arbeitung der Zeichen.
;=====
;
    .MODEL SMALL
    .RADIX 16          ; Hexadezimalsystem
    EXTRN HexAsc:NEAR ; Externes Modul

    .CODE
;
;=====
; SKIP Separator (Blank, Komma)
;
; Aufgabe: Suche die Separatoren Blank oder
;         Komma und überlese sie.
;
; CALL: ES:DI -> Adresse ParameterString
;       CX     Länge Parameterstring
; Ret.: CY : 0 o.k.
;       ES:DI -> Adresse 1. Zchn. Parameter
;       CY : 1 Ende Parameterliste erreicht
;=====
;
    PUBLIC Skip

```

```

;
Skip   PROC NEAR
Loops: CMP CX,0000      ; Ende Parameterliste ?
      JNZ Test1        ; Nein -> JMP Test
      STC              ; markiere Ende mit Carry
      JMP SHORT Exit2  ; JMP Exit2
Test1: CMP BYTE PTR ES:[DI], ' ' ; Zchn. = Blank ?
      JZ  Skip1        ; Ja -> Skip
      CMP BYTE PTR ES:[DI], ',' ; Zchn. = "," ?
      JNZ Exit1       ; Nein -> Exit1
Skip1: DEC CX          ; Count - 1
      INC DI           ; Ptr to next Char.
      JMP NEAR PTR Loops ; JMP Loop
Exit1: CLC             ; Clear Carry
Exit2: RET
Skip   ENDP
;
;-----
; Display String
;
; Aufgabe: Gebe einen String innerhalb der
;          Parameterliste aus.
;
; CALL: ES:DI -> Adresse "....." String
;       CX      Länge Parameterstring
; Ret.: CY : 0   o.k.
;       ES:DI -> Adresse nach String
;       CY : 1  Ende Parameterliste erreicht
;-----
;
      PUBLIC String
;
String PROC NEAR

Begin: CMP BYTE PTR ES:[DI],22 ; " gefunden ?
      JNZ Exit3                ; kein String -> EXIT
      INC DI                   ; auf nächstes Zeichen
      DEC CX                    ; Count - 1
Loop1: CMP CX,0000             ; Ende Parameterliste ?
      JNZ Test2                ; Nein -> JMP Test
      STC                      ; markiere Ende mit Carry
      RET                      ; Exit
Test2: CMP BYTE PTR ES:[DI],22 ; " -> Stringende ?
      JZ  Ende                 ; Ja -> JMP Ende
Write: MOV DL,ES:[DI]         ; lese Zeichen
      MOV AH,02                ; DOS-Code
      INT 21                   ; ausgeben
      DEC CX                    ; Count - 1
      INC DI                   ; Ptr to next Char.
      JMP NEAR PTR Loop1      ; JMP Loop
Ende:  INC DI                  ; auf Stringende
      DEC CX                    ; Count - 1
Exit3: CLC                     ; ok-> clear Carry
      RET
String ENDP
;
;-----
; Display Number
;

```

```

; Aufgabe: Gebe eine HEX-Zahl innerhalb der
;           Parameterliste aus.
;
; CALL: ES:DI -> Adresse 1.Ziffer (XX)
;        CX      Länge Parameterstring
; Ret.: CY : 0   o.k.
;        ES:DI -> Adresse nach Zahl
;        CY : 1  Ende Parameterliste erreicht
;-----
;
;           PUBLIC Number
;
Number  PROC NEAR

        CALL NEAR PTR HexAsc ; 1. Ziffer konvert.
        JC Error             ; keine Ziffer -> Error
        MOV DL,AL            ; merke Ergebnis
; 2. Ziffer lesen
        INC DI                ; nächstes Zeichen
        DEC CX                ; Zähler - 1
        CMP CX,0000          ; Pufferende ?
        JZ Display           ; JMP Display
        CALL NEAR PTR HexAsc ; 2. Ziffer konvert.
        JC Display           ; keine Ziffer -> JMP Display
;
; schiebe 1. Ziffer in High Nibble
;
        PUSH CX               ; schiebe 1. Ziffer
        MOV CL,04             ; in High Nibble
        SHL DL,CL
        POP CX
        OR  DL,AL             ; Low Nibble einblenden
Display:MOV AH,02             ; DOS-Code
        INT 21                ; Code in DL ausgeben
        AND CX,CX             ; Ende Parameterliste?
        STC                   ; set Carry zur Vorsicht
        JZ Exit4              ; Ende erreicht -> Exit
        INC DI                ; auf nächstes Zeichen
        DEC CX                ; Count - 1
        CLC                   ; Clear Carry
Exit4:  RET
;-----
; Ausgabe des Fehlertextes und Exit zu DOS
;-----
Error:  MOV AX,SEG Txt        ; DS: auf Text
        MOV DS,AX
        MOV AH,09            ; DOS-Code
        MOV DX,OFFSET Txt    ; Adr. String
        INT 21                ; Ausgabe
        MOV AX,4C01          ; DOS-Code
        INT 21                ; Exit
;
;=====
; Fehlertext
;=====
Txt BYTE "Fehler in der Parameterliste",0D,0A,"$"
;
Number ENDP
        END

```

Listing 5.6: MODULE.ASM

Damit wird das eigentliche Hauptprogramm sehr kurz. Es muß lediglich die Adresse der DOS-Kommandozeile ermitteln und dann die Parameter auswerten. Zur Auswertung werden dann die Unterprogramme benutzt. Das Programm besitzt folgenden Aufbau:

```

;=====
; File : ESC.ASM (c) G. Born
; Version: 1.0 (MASM 6.x)
; Programm zur Ausgabe von Zeichen an die
; Standardausgabeeinheit. Das Programm wird
; z.B. mit: ESC 0D,0A,1B "Hallo"
; von der DOS-Kommandoebene aufgerufen und
; gibt die spezifizierten Codes aus.
;=====
.MODEL SMALL
.RADIX 16          ; Hexadezimalsystem

Blank EQU 20      ; Blank

EXTERN HexAsc:NEAR ; externe Module
EXTERN String:NEAR
EXTERN Number:NEAR
EXTERN Skip:NEAR

;=====
; Definition des Stacksegmentes
;=====
.STACK
.CODE

;
;=====
; Hauptprogramm
;=====
;
Start: MOV AH,51      ; ermittle Adr. PSP
      INT 21         ; "
      MOV ES,BX      ; ES = Adr. PSP !
      MOV CL,ES:[80] ; lade Pufferlänge
      CMP CL,0       ; Text vorhanden
      JZ Endx        ; kein Text vorhanden
      XOR CH,CH      ; clear Counter Hbyte
      MOV DI,0081    ; lade Puffer Offset
Loopb:                ; gebe Parameter aus
      AND CX,CX      ; Ende erreicht ?
      JZ Endx        ; DOS-Exit
      CALL NEAR PTR Skip ; CALL Skip
      JC Endx        ; DOS-Exit
      CALL NEAR PTR String; CALL String
      JC Endx        ; DOS-Exit
      CALL NEAR PTR Skip ; CALL Skip
      JC Endx        ; DOS-Exit
      CALL NEAR PTR Number; CALL Number
      JC Endx        ; DOS-Exit

```

```
JMP SHORT Loopb ; JMP Loop
;
; DOS-Exit, Returncode in AL
;
Endx:  MOV AX,4C00      ; DOS Exitcode
        INT 21
        END Start
```

*Listing 5.7: ESC.ASM*

Die einzelnen Quelldateien sind mit den Anweisungen:

```
ML /FL ESC.ASM
ML /FL MODULE.ASM
ML /FI HEXASC.ASM
```

zu übersetzen. Der Macroassembler wird zwar einige Fehlermeldungen bringen. So signalisiert er zum Beispiel, daß in ASK verschiedene Prozeduren (Skip, etc.) aufgerufen werden, die nicht in der Quelldatei vorhanden sind. Dies ist nicht weiter tragisch, daß anschließend die einzelnen OBJ-Files mit LINK kombiniert werden:

```
LINK ESC+MODULE+HEXASC,,,,,
```

Damit werden die OBJ-Files zu einem lauffähigen EXE-Programm kombiniert. Der Linker löst auch die vom Assembler gemeldeten externen Referenzen auf. Sofern der Linker keine Fehlermeldungen mehr bringt kann nun das Programm ESC.EXE mit einem Debugger getestet werden.

## 5.12.2 Die Directiven für externe Module

In obigen Quellprogramme finden sich einige neue Anweisungen für den Assembler die kurz vorstellen möchte. Die nachfolgenden Directiven beziehen sich auf getrennt zu assemblierende Module und sind für den Linker erforderlich.

### Die PUBLIC-Directive

Diese Anweisung erlaubt die explizite Auflistung von Symbolen (Variable, Prozeduren, etc.), die durch andere Module adressierbar sein sollen. Der Linker wertet diese Informationen aus den .OBJ-Dateien aus und verknüpft offene Verweise auf diese Symbole mit den entsprechenden Adressen. So kann zu einem Programm eine Prozedur aus einer fremden Objektdatei zugebunden werden. Der Linker setzt dann im CALL-Aufruf die Adresse des Unterprogrammes ein. Weiterhin lassen sich mit der PUBLIC-Directive Variable als global definieren, so daß andere Module darauf zugreifen können. Die Anweisung sollte im Modulkopf stehen und kann folgende Form besitzen:

PUBLIC Skip, Number, String

Die Definitionen dürfen aber auch direkt vor dem jeweiligen Unterprogramm angegeben werden.

Die Symbole Skip, String und Number lassen sich dann aus anderen Modulen (hier aus ESC) ansprechen, wenn sie dort als EXTRN erklärt werden. Fehlt die PUBLIC-Anweisung, wird der Linker eine Fehlermeldung mit den offenen Referenzen angeben. Zur Erhöhung der Programtransparenz und zur Vermeidung von Seiteneffekten sollten nur die globalen Symbole in der PUBLIC-Anweisungsliste aufgeführt werden.

### Die EXTRN-Directive

Diese Directive ist das Gegenstück zur PUBLIC-Directive. Soll in einem Programm auf ein Unterprogramm oder eine Variable aus einem anderen .OBJ-File zugegriffen werden, muß dies dem MASM bekannt sein. Mit der EXTRN-Directive nimmt der MASM an, daß das Symbol in einer externen .OBJ-Datei abgelegt wurde und durch den Linker überprüft wird. Die EXTRN-Directive besitzt folgendes Format:

```
EXTRN Name1:Typ, Name2:Typ, ....  
EXTERN Name:Typ
```

Als Name ist der entsprechende Symbolname einzutragen. Weiterhin muß der Typ der Referenz in der Deklaration angegeben werden. Hierfür gilt:

```
BYTE oder NEAR  
WORD oder ABS  
DWORD  
QWORD  
FAR
```

Das Gegenstück zu obiger PUBLIC-Definition ist z.B. die Erklärung der externen Referenzen im Hauptprogramm:

```
EXTRN Skip:NEAR, String:NEAR, Number:NEAR
```

Tritt nun eine Referenz auf ein solches Symbol auf (z.B: CALL NEAR PTR Skip), generiert der MASM einen Unterprogrammaufruf ohne die absolute Adresse einzutragen. Die Adresse wird beim LINK-Aufruf dann nachgetragen.

### Die END- Directive

Diese Directive signalisiert das Ende des Assemblermoduls. Die Anweisung besitzt die Form:

```
END  
END start_adr
```

wobei der Name *start\_adr* als Label für Referenzen dienen darf. MASM behandelt den hinter END angegebenen Namen wie ein Symbol, welches mit EQU definiert wurde. So läßt sich dann die Programmlänge leicht ermitteln. Folgen weitere Anweisungen, werden diese als getrenntes Programm übersetzt. Damit lassen sich in einer Quellcodedatei mehrere Module ablegen und mit einem Durchlauf assemblieren.

## Die GROUP-Directive

Mit dieser Directive wird der Linker angewiesen, alle angegebenen Programmsegmente innerhalb eines 64-KByte-Blocks zu kombinieren. Es gilt dabei die Syntax.

```
Group_name GROUP Seg_name1, Seg_name2, ...
```

Der Group\_name läßt sich dann innerhalb der Module der Gruppe verwenden (z.B.: MOV AX,Group\_name). Passen die Module nicht in einen 64-KByte-Block, gibt der Linker eine Fehlermeldung aus. Mit dieser Anweisung lassen sich einzelne Module zu Gruppen kombinieren und in einem Segment ablegen. Zu Beginn des Segmentes werden die Segmentregister gesetzt und sind für alle Module der Gruppe gültig. Wird die GROUP-Directive verwendet, muß sie allerdings in allen Modulen benutzt werden, da sonst einzelne Module nicht im Block kombiniert werden. Die GROUP-Anweisung wird in den Beispielprogrammen nicht benutzt.

## 5.13 Einbinden von Assemblerprogrammen in Hochsprachen

Abschließend möchte ich noch kurz auf die Einbindung eines Assemblerprogramms in Hochsprachen eingehen. Hierzu ist das Modul wie in obigem Beispiel als Prozedur in einer Quelldatei zu speichern und durch den MASM in eine OBJ-Datei zu übersetzen. Dann kann das Modul per LINK in eine Hochsprachenapplikation eingebunden werden.

### Programmbeispiel

Nachfolgend möchte ich kurz an Hand eines kleinen Beispiels die Einbindung von Assemblerprogrammen in Turbo Pascal- und BASIC-Programme zeigen.

Aufgabe ist es eine Prüfsumme nach dem CRC16-Verfahren zu berechnen. Diese Operation wird bei der Datenübertragung zur Fehlerprüfung häufig verwendet. Die Realisierung einer CRC-Berechnung ist in Hochsprachen nicht effizient möglich.

Deshalb wird die Routine in Assembler kodiert und als OBJ-File in die Hochsprachenapplikation eingebunden.

Das Programm CRC.ASM übernimmt die Berechnung der CRC-Summe. Die Parameter werden per Stack übergeben. Das Modul ist als OBJ-File zu übersetzen. Der genaue Aufbau der Parameterübergabe und des Programmes ist nachfolgendem Listing zu entnehmen.

```

;-----
; File       : CRC.ASM
; Version    : 1.1 (MASM 6.x)
; Last Edit  : 12.9.91
; Funktion   :
;
;           CRC 16 Generator für den 8086 Prozessor
;
; Es wird das Polynom  $y = x^{16} + x^{15} + x^2 + 1$ 
; verwendet
; Aufruf von Hochsprachen mit:
;
; CALL CRC (CRCr, Buff, Len)
;
; CRCr = Adresse Variable 16 Bit CRC Register
; Buff = Adresse des Zeichenpuffers
; Len  = Wert Zeichenzahl im Puffer
;
; Es sind die Adressen der zwei Parameter CRCr, Buff und
; der Wert von Len über den Stack zu übergeben:
;
;           Stack Anordnung
;           +-----+
;           ! Seg:   CRCr       !
;           +-----+ + 0E
;           ! Ofs:   CRCr       !
;           +-----+ + 0C
;           ! Seg:   Buff       !
;           +-----+ + 0A
;           ! Ofs:   Buff       !
;           +-----+ + 08
;           ! Wert   Len        !
;           +-----+ + 06
;           ! Seg:   Return Adr.!
;           +-----+ + 04
;           ! Ofs:   Return Adr.!
;           +-----+ + 02
;           ! BP   von CRC      !
;           +-----+
;
; Die Procedur ist als FAR aufzurufen. Es wird
; angenommen, daß die Variablen im DS-Segment
; liegen. Diese Konvention entspricht der
; Parameterübergabe in Turbo Pascal und Quick
; Basic. Für die eigentliche CRC-Berechnung
; gilt folgende Registerbelegung:
;
; Register: BX = CRC Register
;           SI = Zeiger in den Datenstrom

```

```

;          CX = Zahl der Daten
;          AH = Bit Counter
;          AL = Hilfsaccu
;
;-----
;
;          .RADIX      16
;          .MODEL      LARGE
POLY      EQU          0010000000000001y ; Polynom
;
;          .CODE
CRC       PUBLIC      CRC
          PROC        FAR
CRCX:     PUSH        BP          ; save old frame
          MOV         BP,SP      ; set new frame
          PUSH       DS          ; save DS
          MOV        CX,[BP]+06  ; get (Len)
          MOV        SI,[BP]+08  ; get Adr (Buff)
          MOV        DS,[BP]+0A  ; get Seg (Buff)
          MOV        BX,[BP]+0C  ; get Adr (CRC)
          MOV        DI,BX       ; merke Adr
          MOV        BX,DS:[BX]  ; load CRC value
;
          CLD                    ; Autoincrement
;
LESE:     MOV         AH,08       ; 8 Bit pro Zchn
          LODSB        ; get Zchn & SI+1
          MOV         DL, AL      ; merke Zeichen
;
; CRC Generator
;
CRC1:     XOR         AL,BL       ; BCC-LSB XOR Zchn
          RCR         AL,1       ; Ergeb. in Carry
          JNC         NULL       ; Serial Quotient?
;
; Serial Quotient = 1
;
EINS:     RCR         BX,1       ; SHIFT CRC right
          XOR         BX,POLY    ; Übertrag
          JMP         TESTE      ; weitere Bits
;
; Serial Quotient = 0
;
NULL:     RCR         BX,1       ; Shift CRC right
;
TESTE:    MOV         AL,DL      ; Lese Zeichen
          RCR         AL,1       ; Zchn 1 Bit right
          MOV        DL,AL      ; merke Rest Zchn
          DEC         AH         ; 8 Bit fertig ?
          JNZ        CRC1       ; Zchn bearbeiten?
          LOOP       LESE       ; String read?
;
; Ende -> Das CRC Ergebnis steht im Register BX
; -> schreibe in Ergebnisvariable zurück
;
          MOV        DS:[DI],BX ; store CRC result

```

	POP	DS	; restore Seg Reg.
	POP	BP	; store old frame
	RETF	0A	; POP 10 Bytes
CRC	ENDP		
	END		

Listing 5.8: CRC.ASM

Die Einbindung des OBJ-Moduls in Turbo Pascal (z.B. Version 4.x- 7.x) ist dann recht einfach. Wichtig ist, daß die Prozedur als FAR aufgerufen wird und die Übergabeparameter korrekt definiert wurden. Das nachfolgendes Listing zeigt beispielhaft wie dies realisiert werden kann.

```
{*****}
File      : DEMO.PAS
Vers.    : 1.1
Autor    : G. Born
Files    : ---
Progr. Spr.: Turbo Pascal 4.0 (und höher)
Betr. Sys.: DOS ab 2.1
Funktion: Das Programm dient zur Demonstration
des Aufrufes von Assemblerprogrammen aus Turbo
Pascal. Es wird das Programm CRC.OBJ einge-
bunden. Die zu übertragenden Zeichen stehen als
Bytes im Feld buff[.
*****}

TYPE Buffer = Array [1 .. 255] OF Byte;

VAR crc_res : Word;           { CRC Register }
    buff : Buffer;           { Datenpuffer }

{*****      Hilfroutinen      *****}

{*
  Hier wird die OBJ-Datei eingebunden
  und die Prozedur definiert
*}
{$L CRC.OBJ}                 { OBJ. File      }
{$F+}                        { FAR Modell ! }
procedure CRC (var crc_res : word; var buff : Buffer; len :
integer);
external;
{$F-}

procedure Write_hex (value, len : integer);
{
  Ausgabe eines Wertes als Zahl auf dem Bildschirm.
  Durch Len wird festgelegt, ob ein Byte (Len = 1)
  oder Wort (Len = 2) ausgegeben werden soll.
}
const Hexzif : array [0..15] of char = '0123456789ABCDEF';
      Byte_len = 1;
      Word_len = 2;
```

```

TYPE zahl = 1..2;
VAR temp : integer;
    carry : zahl;
    i     : zahl;
begin
  if len = Word_len then
    begin
      temp := swap (value) and $0FF;      { high byte holen    }
      write (Hexzif[temp div 16]:1,Hexzif[temp mod 16]:1);
      end;
      temp := value and $0FF;            { low byte holen     }
      write (Hexzif[temp div 16]:1,Hexzif[temp mod 16]:1);
    end; { Write_hex }

{**** Hauptprogramm ****}

begin

  crc_res := 0;                          { clear CRC - Register }
  buff[1] := $55;                         { Testcode setzen     }
  buff[2] := $88;
  buff[3] := $CC;

  writeln ('CRC - Demo (c) Born G. ');
  writeln;
  writeln ('CRC-Berechnung per Polynomdivision');
  writeln;

  CRC (crc_res, buff, 3);                  { Aufruf CRC Routine 1 }
  write ('Die CRC - Summe ist : ');
  write_hex (crc_res,2);                  { Hexzahl ausgeben   }
  writeln;
end.                                       { Ende                 }

```

Listing 5.9: DEMO.PAS

Anschließend läßt sich das Programm als EXE-File mit der Eingabe:

DEMO

aufrufen.

Um das Programm CRC.OBJ in QuickBasic 4.x einzubinden, geht man analog vor. Das nachfolgende Listing in QuickBasic demonstriert die Einbindung des OBJ-Moduls.

```

'|*****
'| File       : DEMO.BAS
'| Vers.     : 1.1
'| Autor      : G. Born
'| Files     : ---
'| Progr. Spr.: Quick Basic 4.x
'| Betr. Sys. : DOS ab 2.1
'| Funktion: Das Programm dient zur Demonstration der Ein-

```

```

'!      bindung von Assemblermodulen in QuickBasic.
'!      Es wird das Modul CRC.OBJ benutzt, Die Parameter
'!      stehen als Bytes im Feld buff%(), oder im String
'!      buff$. Aus diesen Zeichen wird dann die CRC16-
'!      Summe mittels der Proedur CRC (File CRC.OBJ)
'!      berechnet. Compiler und Linker sind mit folgenden
'!      Parametern aufzurufen:
'!
'!      BC DEMO.BAS
'!      LINK DEMO.OBJ,CRCA.OBJ
'!
'|*****
DIM buff%(255)           '! Integer Puffer

crcres% = 0              '! clear CRC-Register
'!
'! setze Zeichen in Puffer, beachte aber, daß es kein Byte
'! Datum gibt, d.h. zwei Bytes sind in einer Integer Variablen
'! zu speichern !!!
'!
buff%(0) = &H8855        '! setze Zeichen in
buff%(1) = &HCC          '! INTEGER Puffer
'!
'! Setze Zeichen alternativ in den Stingpuffer
'!
buff$ = CHR$(&H55)+CHR$(&H88)+CHR$(&HCC) '! Testcode setzen

PRINT "CRC-Demo Programm in Basic (c) Born G."
PRINT
PRINT "CRC-Berechnung per Polynomdivision"
PRINT
'!
'! Berechne CRC aus Integer Puffer
'!
CALL CRC (SEG crcres%, SEG buff%(0), BYVAL 3) '! Aufruf CRC
Routine 1

PRINT "Die CRC - Summe ist : ";HEX$(crcres%) '! Hexzahl ausgeben

'!
'! Berechne CRC aus String Puffer
'!
crcres% = 0              '! clear CRC-Register
FOR i% = 1 TO LEN(buff$) '! separiere Zeichen
  tmp% = ASC(MID$(buff$,i%,1)) '! in Integer wandeln
  CALL CRC (SEG crcres%, SEG tmp%, BYVAL 1) '! CRC Routine 1
NEXT i%

PRINT
'! Hexzahl ausgeben
PRINT "Die CRC - Summe ist : ";HEX$(crcres%)
END
'|**** Ende ****

```

Listing 5.10: DEMO.BAS

Die Einbindung in andere Programmiersprachen kann analog erfolgen. Gegebenenfalls sind die Compilerhandbücher als Referenz zu benutzen.

Damit möchte ich die Ausführungen zu MASM beenden. Sicherlich konnten nicht alle Aspekte des Produkts vorgestellt werden. Hier ist die entsprechende Originaldokumentation des Herstellers zu konsultieren. Für einen ersten Einstieg sollte der vorliegende Text aber genügen.



## 6 Programmentwicklung mit TASM

In Kapitel 2 wurden im wesentlichen die Assembleranweisungen des 8086-Prozessors vorgestellt. Die Programme ließen sich mittels DEBUG in COM-Dateien umwandeln. Deshalb wird man nach der Einarbeitung in den 8086-Befehlssatz meist auf einen Assembler umsteigen. Leider sind die Steueranweisungen zwischen A86, MASM und den Borland Turbo Assemblern (TASM) nicht ganz kompatibel. Zusätzlich enthalten die Borland Hochsprachen die Möglichkeit, Inline-Assembleranweisungen zu verarbeiten. Im vorliegenden Kapitel möchte ich auf die Steueranweisungen des Borland Turbo Assembler (TASM bis zur Version 3.0) gehen. Diese Einführung kann sicherlich nicht als Ersatz für die Originaldokumentation dienen. Besitzer des TASM steht aber ein umfangreicher Handbuchsatz zur Verfügung. Weiterhin gibt es spezielle Literatur zum TASM. Aber für die ersten Schritte sollten die folgenden Ausführungen eine gute Hilfe sein - was letztlich das Ziel dieses Buches ist.

Zur Steuerung des Assemblers müssen eine Reihe von Pseudobefehlen im Programm angegeben werden. Diese Befehle generieren keinen Code, sondern beeinflussen lediglich den Übersetzervorgang. Nachfolgend werde ich kurz auf diese Pseudobefehle (Directiven) eingehen. Weitere Hinweise finden sich in der Originaldokumentation des TASM.

Ein einfaches Assemblerprogramm zur Ausgabe des Textes *Hallo* besitzt dann folgenden Aufbau:

```
=====
; File:      HELLO.ASM (c) Born G.
; Version:  2.0 (TASM 2.0/3.0)
; Aufgabe:  Programm zur Ausgabe der
; Meldung:  "HALLO" auf dem Bildschirm.
; Das Programm ist als COM-Datei mit
; TASM zu übersetzen!!
=====
;
;          .MODEL TINY          ; COM-Datei
;          .RADIX 16           ; Hexadezimalsystem
;          .CODE
;          ORG 0100            ; Startadresse COM
;
HALLO:    MOV AH,09             ; DOS-Display Code
;          MOV DX,OFFSET TEXT  ; Textadresse
;          INT 21              ; DOS-Ausgabe
;
;          MOV AX,4C00         ; DOS-Exit Code
;          INT 21              ; terminiere
;
```

```

; Textstring als Konstante
;
TEXT DB "Hallo", 0A, 0D, "$"
;
END Hallo

```

Listing 6.1: HELLO.ASM im TASM-Format

Auch dieses Listing offenbart, daß die Sache wesentlich komplexer als mit DEBUG ist. Der TASM erwartet eine Reihe von zusätzlichen Steueranweisungen, deren Sinn nicht intuitiv klar wird. Um das Programm möglichst einfach zu halten, wurde hier noch das COM-File-Format verwendet, in welches die ASM-Datei zu übersetzen ist. Dieses COM-Format stammt noch aus der CP/M-Zeit und begrenzt die Größe eines Programmes auf 64 KByte. Dies ist bei unseren kleinen Programmen sicherlich kein Handikap. Vorteile bringt allerdings die Tatsache, daß DOS beim Laden eines COM-Programms alle Segmentregister mit dem gleichen Wert initialisiert (Bild 6.1).

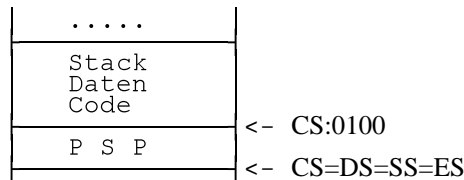


Bild 6.1: Segmentbelegung bei COM-Programmen

Ab der Adresse CS:100 beginnt der Codebereich mit den Programmanweisungen. Im gleichen Segment befinden sich auch die Daten und der Stack. Beim Zugriff auf Daten kann das Programm recht einfach aufgebaut werden. Um ein COM-Programm zu erstellen, ist obige Quelldatei mit dem Namen HELLO.ASM zu erzeugen und mit dem Befehl:

```
TASM HELLO.ASM,,,
```

zu übersetzen. Die Kommas hinter dem Namen der Quelldatei signalisieren TASM, daß die OBJ- und Listdateien mit dem gleichen Namen, ergänzt um die entsprechende Extension, zu speichern sind. Nach dem fehlerfreien Durchlauf durch TASM muß die entstehende OBJ-Datei gelinkt werden. Die ist mit der Anweisung:

```
TLINK HELLO.OBJ /t
```

möglich. Der Schalter /T sorgt dafür, daß ein COM-Programm beim Linken erzeugt wird. Sobald das Programm fehlerfrei übersetzt und gelinkt wurde, liegt eine COM-Datei vor, die sich unter DOS ausführen läßt.

Obiges Listing enthält aber bereits eine Reihe von Steueranweisungen (Directiven), die ich nun kurz erläutern möchte.

## 6.1 Die Darstellung von Konstanten

Im Assemblerprogramm lassen sich Konstante innerhalb von Anweisungen (z.B. MOV AX,3FFF) eingeben. Dabei dürfen die Konstanten in verschiedenen Zahlensystemen eingegeben werden. Der TASM benutzt einige implizite Konventionen:

- ◆ Standardmäßig werden Zahlen als Dezimalwerte interpretiert (z.B. 21).
- ◆ Wird an eine Zahl der Buchstabe h angefügt, interpretiert TASM diese Zahl als Hexadezimalwert (z.B. 3FFFh). Eine Hexzahl muß immer mit den Ziffern 0..9 beginnen. Falls die erste Ziffer zwischen A und F liegt, ist eine 0 voranzustellen (z.B. C000H wird zu 0C000H).
- ◆ Binärzahlen sind durch einen angehängten Buchstaben b zu markieren. Dann sind nur die Ziffern 0 und 1 innerhalb der Zahl erlaubt.
- ◆ Dezimalzahlen lassen sich explizit durch ein angehängtes t (z.B: 100t) darstellen.
- ◆ Mit dem Buchstaben o hinter der letzten Ziffer werden Oktalzahlen markiert.

Die Buchstaben für die Zahlenbasis können sowohl in Groß- als auch in Kleinbuchstaben angegeben werden.

## 6.2 Die RADIX-Directive

Die obigen impliziten Definitionen für die Interpretation der Zahlenbasis ist mit zusätzlichem Aufwand verbunden. Deshalb verzichten viele Programmierer auf den angehängten Buchstaben zur Markierung der Zahlenbasis. Hier läßt sich mit dem RADIX-Kommando die eingestellte Zahlenbasis überschreiben. Die Anweisung besteht aus dem Schlüsselwort und einer Dezimalzahl, die die gewünschte Zahlenbasis spezifiziert: RADIX xx. Die folgenden Anweisungen verdeutlichen die Anwendung des Befehls:

```
RADIX 10 ; alle Werte als Dezimalzahlen lesen  
RADIX 16 ; alle Werte als Hexadezimalzahlen  
RADIX 2 ; alle Werte als Binärzahlen lesen
```

Dem RADIX-Kommando ist ein Punkt voranzustellen. Ich habe mir angewöhnt in allen Assemblerprogrammen mit Hexadezimalzahlen zu arbeiten. Deshalb steht die RADIX-Directive auch zu Beginn eines jeden Programmes, um die Zahlenbasis 16 zu vereinbaren.

## 6.3 Definition von Datenbereichen

Häufig möchte man Variable innerhalb eines Programms anlegen. Dabei ist für jeden Eintrag genügend Speicher vorzusehen.

### 6.3.1 Datendefinitionen mit DB, DW, DD, DQ und DT

Zur Definition von Variablen kennt der TASM - wie MASM - die Pseudoanweisungen:

DB (erzeugt ein Data Byte)  
DW (erzeugt ein Data Word)  
DD (erzeugt ein Double Data Word = 4 Byte)  
DF,DP (erzeugt ein FAR Data Word = 6 Byte)  
DQ (erzeugt ein Quad Data Word = 8 Byte)  
DT (erzeugt 10 Data Byte)

Die DQ-Anweisung erzeugt 8 Byte, welche zur Speicherung von 8087-Werten gebraucht werden. Die DT-Anweisung erzeugt eine Variable mit 10 Byte. Vor dem Pseudobefehl kann der Name der Variablen stehen:

Zahlen DB 1,2,3,4,5,6  
Wort DW OFFFFH

Über den Namen läßt sich später auf die Daten zurückgreifen (z.B. MOV DX,OFFSET Text). Beachten Sie, daß hinter dem Variablen kein Doppelpunkt steht.

Soll ein Textstring definiert werden, kann dies in älteren TASM-Versionen (und zumindest bei den meisten Assemblern) mit dem Befehl:

String DB "Dies ist ein String", 0A, 0D,"\$"

erfolgen. Doch nun zurück zu unserem ersten TASM-Programm. Mit dem Befehl:

TEXT DB "Hallo", 0A, 0D,"\$"

wird ein Datenbereich, bestehend aus mehreren Bytes vereinbart, der sich zusätzlich über den Namen Text ansprechen läßt. Der Assembler initialisiert den Text dann mit den angegebenen Startwerten. Konstante, die mit Werten initialisiert werden, sollten im Codesegment gespeichert werden. Variable, deren Inhalt vom Programm verändert wird, sind dagegen im Datensegment abzulegen. Bezüglich der Definition der Segmente sei auf den Abschnitt *Die Segment Anweisung* und die folgenden Beispielprogramme verwiesen.

## 6.4 Der DUP-Operator

Um größere Datenbereiche mit dem gleichen Wert zu initialisieren, existiert das Schlüsselwort DUP. Soll zum Beispiel ein Puffer mit 5 Byte angelegt werden, müßte die DB-Directive folgendes Aussehen haben:

```
Buffer DB 00, 00, 00, 00, 00
```

Bei diesem Beispiel läßt sich die Definition noch leicht als Quellzeile formulieren. Was ist aber, falls der Puffer 256 Zeichen umfassen soll? Es macht wohl keinen Sinn hier 256 Byte mit dem Wert 0 im Quellprogramm aufzunehmen. Die meisten Assembler bieten aber die Möglichkeit einen Datenbereich mit n Byte zu reservieren und gegebenenfalls mit Startwerten zu belegen. Um einen Puffer mit 32 Byte Länge zu vereinbaren ist folgende Anweisung möglich:

```
Buffer DB 20 DUP (0)
```

Beachten Sie dabei, daß die Längenangabe hier als Hexzahl (20H = 32) erfolgt. Der Puffer wird mit den Werten 00H initialisiert. Um alle Bytes auf den Wert FFH zu setzen, wäre die Anweisung:

```
Buffer DB 20 DUP (FF)
```

erforderlich. Das bedeutet:

- ◆ Die Zahl vor dem DUP-Operator gibt die Zahl der zu wiederholenden Elemente wieder. Beim DB-Operator sind dies n Byte, beim WORD-Operator n Worte.
- ◆ Nach dem DUP-Operator folgt der Initialisierungswert in Klammern.

Soll eine Variable nicht initialisiert werden, ist an Stelle des Wertes ein Fragezeichen (?) einzusetzen.

```
Buffer DB 20 DUP (?)
```

Der Assembler reserviert dann einen entsprechenden Speicherbereich für die Variable.

ASCII-Strings lassen sich einfach in Hochkommas einschließen und im Anschluß an eine DB-Anweisung eingeben:

```
Text DB 'Die ist ein Text',0D,0A
```

In obigem Beispiel werden Textzeichen und Hexbytes gemischt. Zur Wahrung der Kompatibilität dürfen für Textkonstante auch Hochkommas verwendet werden. Es ist

aber auf die unterschiedliche Speicherung von Bytes und Worten zu achten. Die Anweisungen:

```
Text1 DB 'AB' (speichert LOW 41H, High 42H)
Text2 DW 'AB' (speichert 42H, 41H)
```

legen die Bytes in unterschiedlichen Speicherzellen ab. Im ersten Fall steht der Wert 'A' auf dem unteren Byte, daran schließt sich der Buchstabe 'B' an. Bei der Abspeicherung von Worten ist der linke Buchstabe dem höheren Byte zugeordnet. Mit der Anweisung:

```
Zeiger DW FFFF0000
```

lassen sich 32-Bit-Werte (FAR Pointer) definieren.

Der DUP-Operator läßt sich übrigens auch schachteln. Die Anweisung:

```
Buffer DB 5 DUP ( 5 DUP (5 DUP (1)))
```

legt einen Puffer von  $5 * 5 * 5$  (also 125 Byte) an, der mit den Werten 01H gefüllt wird.

**Achtung:** Beachten Sie aber, daß der TASM standardmäßig alle Werte als Dezimalzahlen interpretiert. Die Angaben in einer DUP-Anweisung, oder der Initialisierungswert, werden dann als Dezimalzahl ausgewertet.

```
Buffer 256 DUP (23)
```

Legt einen Puffer von 256 Byte an, der mit dem Wert 23 (dezimal) gefüllt wird. Da in der Regel bei der Assemblerprogrammierung mit Hexadezimalwerten gearbeitet wird (denken Sie nur an den INT 21 (hexadezimal)), verwende ich in meinen Programmen die RADIX-Anweisung zur Einstellung der Zahlenbasis 16. Wird dies vergessen, kommt es schnell zu Fehlern. Wer gibt schon in seinen Programmen den Befehl *INT 21H* ein. Wird die RADIX-Directive nicht verwendet, müßte der DOS-INT 21-Aufruf mit *INT 33* (dezimal) kodiert werden.

### 6.4.1 Der LENGTHOF-Operator

Häufig ist es innerhalb eines Assemblerprogramms erforderlich die Länge einer Datenstruktur zu ermitteln. Wurde ein String zum Beispiel mit:

```
Text DB "Hallo dies ist ein Text"
```

vereinbart, läßt sich dieser Text Byte für Byte bearbeiten. Der TASM bietet keine Directive LENGTHOF wie MASM zur Abfrage der Zahl der Elemente der Variablen. Mit:

```
Text DB "Hallo dies ist ein Text"  
Ende LABEL WORD  
LENGTHOF DW (Ende - Text)
```

wird jedoch eine Pseudokonstante `LENGTHOF` definiert, in die TASM die Stringlänge ablegt. Mit:

```
MOV CX, LENGTHOF Text
```

wird der Wert dann als Konstante dem Register `CX` zugewiesen. Der Befehl ist im Prinzip als `MOV CX,23T` zu interpretieren, wobei der Assembler automatisch den Wert `23T` einsetzt.

## 6.4.2 Die Definition von Strukturen

Mit der Anweisung `STRUC` lassen sich mehrere Variable oder Konstante zu kompakteren Datenstrukturen zusammenfassen. Die Directive erzwingt, daß die Variable zusammen abgespeichert werden und sich anschließend durch die Zeigerregister `BX`, `BP`, `DI` oder `SI` über das erste Element adressieren lassen. Die indirekte Adressierung über `[BX+SI+Konst.]` eignet sich gut zur Bearbeitung solcher Strukturen. Die Definition:

```
Buffer STRUC  
  Len DB 80          ; Länge Puffer  
  Count DB 00        ; Zeichenzahl  
  Buf DB 80 DUP (?) ; Bufferbereich  
Buffer ENDS
```

legt eine Datenstruktur für einen Puffer an, in der das erste Byte die Pufferlänge enthält. Im zweiten Byte ist die Zahl der im Puffer befindlichen Zeichen definiert. Daran schließt sich ein Pufferbereich von 127 Byte an. Die Definition der Struktur wird mit einer `ENDS`-Directive abgeschlossen.

## 6.4.3 Die MODEL-Directive

Zu Beginn des Programmes läßt sich dem Assembler mitteilen, welches Speichermodell gewählt wurde. Abgeleitet von den 8086-Speicherstruktur existierten verschiedene Optionen:

- ◆ `TINY` veranlaßt, daß Code, Daten und Stack zusammen in einem 64-KByte-Block vereint werden. Dies ist bei `COM`-Programmen erforderlich. Code und Zugriffe auf Daten werden dabei mit `NEAR`-Adresszeigern ausgeführt.

- ◆ SMALL bewirkt, daß ein 64-KByte-Codesegment und ein Datensegment gleicher Größe definiert wird. Dies setzt aber voraus, daß das Programm als EXE-File gelinkt wird.
- ◆ MEDIUM unterstützt mehrere Codesegmente und ein 64 KByte großes Datensegment. Bei dem COMPACT-Modell werden dagegen mehrere Datensegmente, aber nur ein 64-KByte-Codesegment zugelassen.
- ◆ LARGE erlaubt mehrere Daten- und mehrere Codesegmente. Hier erfolgen die Zugriffe auf Code und Daten grundsätzlich mit FAR-Zeigern.

Der TASM unterstützt weitere MODEL-Vereinbarungen, auf die ich an dieser Stelle aber nicht eingehen möchte. Unser Beispielpogramm enthält die TINY-Anweisung, damit sich das Programm in eine COM-Datei verwandeln läßt.

## 6.5 Die Segmentanweisung

Dem Assembler muß ebenfalls mitgeteilt werden, in welchem Segment er die Codes oder die Daten ablegen soll. Ein übersetztes Programm besteht minimal aus einem Code- und einem Datensegment (Ausnahme COM-Programme, die nur ein Segment besitzen). Bei der Erzeugung der .OBJ- und .COM-Files benötigt der TASM zusätzliche Informationen zur Generierung der Segmente (Reihenfolge, Lage, etc.). Hierzu sind mehrere Operatoren zugelassen:

- ◆ CODE markiert den Beginn eines Programmbereiches mit ausführbaren Anweisungen. Neben den Programmanweisungen können hier auch Konstante oder initialisierte Daten abgelegt werden. Interessant ist dies vor allem bei der Erzeugung von .OBJ-Files, da der Linker die Segmente später zusammenfaßt. Im TINY-Modell ist nur ein Codebereich erlaubt.
- ◆ DATA signalisiert, daß ein Datenbereich folgt. Anweisungen wie DB, DW und DQ erzeugen solche Variable. Häufig findet man die Datendefinition zu Beginn eines Assemblerprogramms. Mit der ORG-Anweisung läßt sich die Lage des Datenbereiches explizit festlegen. In den Beispielprogrammen finden sich die Definitionen der Datenbereiche häufig am Programmende hinter dem Code. Dadurch legt der Assembler automatisch die Lage des Datenbereiches korrekt fest. TASM generiert dann Anweisungen, die später im Datensegment des Programmes abgelegt werden. Obiges Beispielprogramm verzichtet auf ein Datensegment, da keine Variablen zu bearbeiten sind.
- ◆ STACK erzeugt ein Segment, welches der Linker für den Stack reserviert. In obigem Beispiel wurde auf den Stack verzichtet, da ein COM-Programm den Stack im oberen Speicherbereich des Segmentes anlegt. Mit .STACK 200H wird automatisch 512 Byte Stack reserviert.

Weitere Beispiele für die Verwendung dieser Segmentdefinitionen finden sich in den folgenden Abschnitten.

### 6.5.1 Die ORG-Anweisung

Mit dieser Anweisung lassen sich Daten und Programmcode an absoluten Adressen im aktuellen Segment speichern. Bei OBJ-Dateien darf die Lage der Code- und Datenbereiche nicht festgelegt werden, da dies Aufgabe des Linkers ist. Bei COM-Programmen müssen dagegen die Adressen im Programm definiert werden. Hierzu dient die ORG-Anweisung, mit der sich die Adresse des nächsten Befehls definieren läßt. Mit `ORG 100` wird TASM angewiesen, die nächste Anweisung ab `CS:0100` zu assemblieren. Der Parameter einer ORG-Anweisung muß entweder eine Konstante oder ein Ausdruck sein. Eine Verwendung eines symbolischen Namens ist ebenfalls möglich, sofern keine Vorwärtsreferenzen dabei notwendig werden. Bei der Assemblierung einer COM-Datei generiert der TASM den Code auch ohne die `ORG 100` Anweisung ab `CS:0100`. In den Beispielprogrammen wird die ORG-Anweisung jedoch verwendet. Allgemein bleibt festzuhalten, daß der ORG-Befehl nur selten im Programm auftreten sollte. Andernfalls besteht die Gefahr, daß sich Codebereiche bei späteren Programmiererweiterungen überlappen und damit zu Fehlern führen.

Die ORG-Anweisungen sind demnach nur bei COM-Programmen erlaubt und sinnvoll. Obiges Programm enthält z.B. eine ORG-Anweisung, so daß der Code ab `CS:100H` beginnt.

## 6.6 Der OFFSET Operator

Beim Zugriff auf Speicherzellen (z.B: `MOV AX, Buffer`) sind zwei Fälle zu unterscheiden. Einmal kann der Inhalt einer Konstanten *Buffer* gemeint sein. Andererseits besteht die Möglichkeit, die Adresse der Variablen *Buffer* in `AX` zu lesen. Um die Konstruktion eindeutig zu gestalten, existiert die Anweisung `OFFSET`, die in den Beispielprogrammen verwendet wird. Mit:

```
MOV AX, OFFSET Buffer
```

wird die (Offset) Adresse der Variablen oder Konstanten *Buffer* durch den Assembler berechnet und in das Register `AX` geladen.

Die Segmentadresse läßt sich übrigens mit dem Operator:

```
MOV BX, SEG Buffer
```

ermitteln.

Um den Inhalt der Variablen Buffer zu laden, ist die indirekte Befehlsform (z.B. MOV AX,[Buffer]) zu nutzen.

### Programmbeispiel

Zur Auflockerung möchte ich an dieser Stelle ein weiteres Programmbeispiel in TASM einfügen. Es handelt sich um das Programm zur Vertauschung der beiden parallelen Schnittstellen LPT1 und LPT2. In TASM besitzt das Programm folgendes Format:

```

;=====
; File: SWAP.ASM   (c) Born G. V 1.0
; Programm zur Vertauschung von LPT1
; und LPT2. Programm als COM-Datei
; mit TASM übersetzen!!
;=====
;
;       .MODEL TINY           ; COM-Datei
;       .RADIX 16             ; Hexadezimalsystem
;       .CODE
;       ORG 0100              ; Startadresse COM
;
SWAP:  MOV AH,09              ; DOS-Display Code
;       MOV DX,OFFSET TEXT    ; Textadresse
;       INT 21                ; DOS-Ausgabe
;
;       MOV AX,0000           ; ES auf Segm. 0000
;       MOV ES,AX             ; setzen
;       MOV AX,ES:[0408]      ; Portadresse LPT1
;       MOV BX,ES:[040A]      ; Portadresse LPT2
;       XCHG AX,BX            ; Swap Adressen
;       MOV ES:[0408],AX      ; store Portadressen
;       MOV ES:[040A],BX      ;
;
;       MOV AX,4C00           ; DOS-Exit Code
;       INT 21                ; terminiere
;
;       .DATA
;
;       ; definiere Textstring im Datenbereich
;
TEXT  DB 'SWAP LPT1 <-> LPT2   (c) Born G.',
;       0D,0A,'$'
;
END Swap

```

*Listing 6.2: SWAP.ASM*

Das Programm wird wieder als COM-Datei übersetzt. Als Erweiterung zur Version aus Kapitel 2 gibt das Programm beim Aufruf noch eine Meldung an den Benutzer aus. Die Textkonstante wird in diesem Beispiel explizit im Datensegment abgelegt. Die Anweisung .DATA definiert den Beginn des Segmentes. Auch hier ist wieder eine Besonderheit bei Verwendung des Segment-Override-Operators zu beachten.

## 6.7 Die Segment-Override-Anweisung

In Kapitel 2 wurde die Adressierung der 8086-Befehle besprochen. Standardmäßig greift der Prozessor auf das Datensegment zu. Bei Adressierung über das BP-Register finden sich die Daten im Stacksegment. Mit der Segment-Override-Anweisung läßt sich die Standardeinstellung für die Segmentierung für den folgenden Befehl aussetzen. In DEBUG muß der Segmentdescriptor vor dem Befehl stehen:

```
ES:MOV AX,[0408] ; Portadresse LPT1
ES:MOV BX,[040A] ; Portadresse LPT2
```

Der TASM generiert bei diesem Konstrukt eine Fehlermeldung, erwartet er doch den Segmentdescriptor vor der eckigen Klammer.

```
MOV AX,ES:[0408] ; Portadresse LPT1
MOV BX,ES:[040A] ; Portadresse LPT2
```

Sollte der Assembler einen Fehler in einer Anweisung mit Segment-Override melden, prüfen Sie bitte zuerst, ob obige Konventionen erfüllt sind.

### Programmbeispiel

Als nächstes Beispiel möchte ich das Programm zur Abschaltung der *NumLock*-Taste in der Version für den TASM vorstellen. Das Programm besitzt folgenden Aufbau:

```

;=====
; File: NUMOFF.ASM (c) Born G.
; Ausschalten der NUM-Lock-Taste.
; Programm als COM-Datei mit TASM
; übersetzen!!
;=====
;
; .MODEL TINY ; COM-Datei
; .RADIX 16 ; Hexadezimalsystem
;
; Programmkonstanten definieren
;
; SEG0 EQU 0000 ; 1. Segment
; Key EQU 0DF ; NumLock Key
; Key_Adr EQU 0417 ; Adresse BIOS
; Exit EQU 4C00 ; DOS-Exit
; DOS_Txt EQU 09 ; DOS-Text
;
; .CODE
; ORG 0100 ; Startadresse COM
;
NUMOFF:MOV AH,DOS_Txt ; DOS-Display Code
MOV DX,OFFSET TEXT ; Textadresse
INT 21 ; DOS-Ausgabe
;

```

```

        MOV AX,SEG0          ; ES auf BIOS-Segm.
        MOV ES,AX           ; setzen
; Bit der NumLock-Taste ausblenden
        AND BYTE PTR ES:[Key_Adr],Key
;
        MOV AX,Exit         ; DOS-Exit Code
        INT 21             ; terminiere
;
        .DATA
;
; definiere Textstring im Datenbereich
TEXT DB 'NUMOFF (c) Born G.',0D,0A,'$'
;
END NUMOFF

```

*Listing 6.3: NUMOFF.ASM*

Auch hier ergibt sich wieder ein neuer Aspekt. Innerhalb des Programmes werden verschiedene Konstanten verwendet (z.B: MOV AH,09 für die Textausgabe). Ändert sich der Wert einer Konstanten, ist gerade bei größeren Programmen die Modifikation sehr fehlerträchtig. Alle Anweisungen, in denen die Konstante vorkommt, müssen gesucht und modifiziert werden. Mit der EQU-Directive läßt sich hier eine bessere Lösung finden.

## 6.8 Die EQU Directive

Mit EQU läßt sich einem Namen eine Konstante, ein Ausdruck, etc. zuweisen. Immer wenn im Verlauf des Programmes dann dieser Name auftaucht, ersetzt der Assembler diesen Namen durch den mit EQU zugewiesenen Wert. Statt der Anweisung: MOV AX,0FFFF läßt sich das Programm wesentlich transparenter mit der Sequenz:

```

True EQU 0FFFF      ; Konstante True
False EQU 0         ; Konstante False
.
MOV AX,True         ; AX Init

```

gestalten. Ähnlich lassen sich Masken, Pufferlängen, etc. mit symbolischen Namen belegen und per EQU definieren. Im Programm taucht dann nur noch der symbolische Name auf. Werden die EQU-Anweisungen im Programmkopf definiert, lassen sich die Programme sehr transparent gestalten, da sich diese EQU's leicht ändern lassen.

In obigem Programmbeispiel wurden alle Konstanten mit EQU am Programmanfang definiert. In der Praxis definiere ich nur die wichtigsten Konstanten im Programmkopf mit EQU. Bei der Anweisung:

```

MOV AH,09
INT 21

```

ist für mich sofort ersichtlich, daß es sich hier um eine Textausgabe unter DOS handelt. Die im Beispiel verwendete Sequenz:

```
NUMOFF:    MOV AH,DOS_Txt    ; DOS-Display Code
           MOV DX,OFFSET TEXT ; Textadresse
           INT 21            ; DOS-Ausgabe
```

ist nach meiner Ansicht nicht so transparent. Letztlich ist es aber Geschmacksache, welche Konstanten direkt im Programm eingesetzt werden und welche global im Programmkopf definiert werden. Zusätzlich lassen sich auch Ausdrücke mit EQU definieren. Die Anweisungen:

```
Len EQU 10T
Step EQU 3
Lang EQU Len * Step
```

weist der Konstanten Lang den Wert 30 (dezimal) zu.

Weiterhin lassen sich die EQU-Definition verwenden um einen Interrupt mit einem symbolischen Namen zu versehen. Der INT 3 wird häufig von Debuggern zum Test verwendet. Die CPU führt im Single-Step-Mode nach jedem Befehl einen INT 3 aus. Es besteht daher die Möglichkeit, den INT 3 durch:

```
TRAP EQU INT 3
```

umzudefinieren. Immer wenn im Programm der Begriff TRAP auftritt, ersetzt der Assembler diesen durch die INT 3-Anweisung.

Mit EQU-Anweisungen kann man in den meisten Assemblern einem Namen durchaus mehrfach verschiedene Werte innerhalb des Programmes zuweisen. Im MASM sind Mehrfachdeklarationen mit EQU unzulässig. Mit dem Operator = kann diese Restriktion allerdings umgangen werden. Er ist als Synonym für EQU zu verwenden. Dann können einem Namen im Verlauf des Programms durchaus unterschiedliche Werte zugewiesen werden.

## Das \$-Symbol

Mit diesem Symbol wird im Assembler die aktuelle Adresse symbolisiert. Um zum Beispiel die Länge einer Stringdefinition zu bestimmen, ist folgende Konstruktion erlaubt:

```
Text DB 'Hallo dies ist ein String'
Len EQU ($ - Text)
```

Der Assembler weist dann der Konstanten *Len* die Länge des Strings *Text* zu.

## 6.9 Operationen auf Ausdrücken

Bei der Erstellung von Assemblerprogrammen werden häufig Konstante in Ausdrücken verwendet (z.B: AND AX,3FFF). Die Konstante ist dabei in der geeigneten Form im Programm anzugeben. Eine Möglichkeit besteht darin, als Programmierer den Wert der Konstanten zu berechnen und im Quellprogramm einzusetzen. Dies ist aber nicht immer erwünscht: so kann es durchaus fehlerträchtig sein, wenn mehrere Werte manuell addiert werden. Um das Programm möglichst transparent zu gestalten, möchte man häufig die ursprünglichen Teilwerte im Programm mit angeben. Die meisten Assembler unterstützen diese Form und berechnen zur Übersetzungszeit den Wert eines Ausdruckes. Der TASM bietet eine Reihe solcher Operatoren, die nachfolgend kurz aufgeführt werden.

### 6.9.1 Addition

Der Operator erlaubt die Addition mehrere Konstante innerhalb eines Ausdruckes. Die Addition kann dabei durch ein + Zeichen markiert werden.

```
K22 EQU 20 + 02
Muster EQU Muster1 + Muster2
MOV AX,04C00 + 22 ; Berechne Konstante
```

Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen, sowie Kommazahlen verwendet werden.

### 6.9.2 Subtraktion

Der Operator erlaubt die Subtraktion von Konstanten innerhalb eines Ausdruckes. Die Subtraktion wird dabei durch das - Zeichen markiert (z.B. MOV AX,033-030). Es dürfen dabei vorzeichenlose und vorzeichenbehaftete Zahlen, sowie Kommazahlen verwendet werden. Bei Variablen muß der Typ der beiden Operatoren übereinstimmen.

### 6.9.3 Multiplikation und Division

Die Operatoren erlaubte die Multiplikation und Division von Konstanten innerhalb eines Ausdruckes. Die Operatoren dürfen nur mit Kommazahlen oder auf ganzen Zahlen durchgeführt werden.

```
CMP CL, 2 * 3 ; Compare mit 6
MOV DX, 256 / 16 ; lade DX mit 16
MOV BX, 4 MOD 2 ; lade BX mit 0
```

Mit Hilfe dieser Operatoren lassen sich Berechnungen zur Assemblierungszeit ausführen. Dies erspart die manuelle (und fehleranfällig) Berechnung durch den Programmierer. Weiterhin wird im Listing sofort sichtbar wie die Konstante berechnet wird.

### 6.9.4 Schiebeoperatoren (SHL, SHR)

Mit den Operatoren SHR und SHL lassen sich Schiebeoperationen auf einer Konstanten oder einem Ausdruck ausführen. Die Operatoren besitzen das Format:

Operand SHR Count (shift right)  
Operand SHL Count (shift left)

Die Schiebefehle erlauben es, den Operanden bitweise nach links oder rechts zu verschieben. Der zweite Operand *Count* gibt dabei die Zahl der zu verschiebenden Binärstellen an. Die Bits, die in den Operanden eingeschoben werden, sind mit dem Wert 0 belegt. Beispiel für die Anwendung der Operatoren sind:

```
MOV BX,0FF33 SHR 4      ; BX = 0FF3  
MOV DX,01 SHL 4        ; DX = 010
```

### 6.9.5 Logische Operatoren

Mit den Anweisungen AND, OR, XOR und NOT lassen sich logische Operationen auf den Operanden ausführen. Die Operatoren besitzen das Format:

Operand AND Operand  
Operand OR Operand  
Operand XOR Operand  
NOT Operand

Die Befehle dürfen ausschließlich mit vorzeichenlosen Byte- oder Word-Konstanten benutzt werden. Die Anweisung:

```
MOV AL,03F AND 0F
```

blendet die oberen 4 Bits der Konstanten 3FH aus. Für die Verknüpfung der Operanden gelten die Regeln für AND, OR, XOR und NOT. Der NOT-Operator invertiert den Wert der angegebenen Konstanten.

## 6.10 Vergleichsoperatoren

Der TASM bietet einen weiteren Set an Vergleichsoperatoren:

Operand EQ Operand (equal)  
Operand NE Operand (not equal)  
Operand LT Operand (less than)  
Operand LE Operand (less or equal)  
Operand GT Operand (greater then)  
Operand GE Operand (greater then or equal)

Als Operanden müssen vorzeichenlose Ganzzahlen (Byte oder Word) angegeben werden. Dabei können sowohl Konstante als auch Variable benutzt werden (z.B. MASKE1 EQ Mode). Als Ergebnis wird ein Byte oder Word zurückgegeben, welches die Werte *true* (0FFFFH) oder *false* (0) enthält. Die Anweisung:

```
MOV AL, 4 EQ 3
```

lädt AL mit dem Wert 0, da der Ausdruck falsch ist.

Die bisher besprochenen Operanden lassen sich in Ausdrücken kombinieren. Dabei gelten die folgenden Prioritäten:

1. Klammern
2. Punkt
3. Segment Override, PTR
4. OFFSET, SEG, TYPE
4. HIGH, LOW
5. +, -
6. \*, /, MOD, SHR, SHL
7. +, - (binär)
8. EQ, NE, LT, LE, GT, GE
9. NOT
10. AND
11. OR, XOR
12. SHORT

bei der Auswertung eines Ausdruckes. Die Zahl 1 definiert dabei die höchste Priorität.

## 6.11 Die EVEN Directive

Mit dieser Anweisung erzwingt der Programmierer, daß der nächste assemblierte Befehl oder die Lage einer Variablen auf einer geraden Speicheradresse festgelegt

wird. Innerhalb des Codesegementes fügt der TASM gegebenenfalls eine NOP-Anweisung ein. Bei Variablen werden diese einfach auf eine gerade Adresse gelegt. Die EVEN-Directive ist besonders bei 80X86-Prozessoren hilfreich, da Zugriffe auf gerade Adressen schneller ausgeführt werden. Ein 16-Bit-Zugriff auf eine ungerade Adresse erfordert 2 Speicheroperationen.

### Programmbeispiel

Doch nun möchte ich ein weiteres Programmbeispiel vorstellen. Erstmals soll nun eine EXE-Datei erstellt werden. Das Programm besitzt den Namen:

ASK.EXE

und erlaubt Benutzerabfragen aus Batchdateien. Das Programm wird mit:

ASK <Text>

aufgerufen. Der Text innerhalb der Aufrufzeile ist dabei optional. Nach dem Start wird der Text der Aufrufzeile am Bildschirm ausgegeben. Dann wird die Tastatur abgefragt und das Zeichen der gedrückten Taste wird an DOS zurückgegeben. Der Tastencode läßt sich innerhalb der Batchdatei dann durch die ERRORLEVEL-Funktion abfragen.

Mit dem Aufruf:

ASK Abbruch (J/N) ?

gibt ASK die Nachricht:

Abbruch (J/N)?

aus und wartet auf eine Benutzereingabe. ASK terminiert sofort nach der Eingabe. Der Code des Zeichens läßt sich in DOS dann per ERRORLEVEL abfragen:

```
ASK Abbruch (J/N)?
IF ERRORLEVEL 75 GOTO NO
IF ERRORLEVEL 74 GOTO YES
:NO .....
```

Zu beachten ist lediglich, daß ERRORLEVEL die Codes zwischen 0 und 255 auf >= prüft. Beispiele für den Einsatz des Programmes ASK finden sich im Anhang und in /2/. Doch nun möchte ich das dazugehörige Listing vorstellen.

```
=====
; File: ASK.ASM    (c) Born G.
; Version: V 1.0  TASM
; Funktion: Programm zur Benutzerab-
```

```

; frage in Batchdateien. Aufruf:
;
;   ASK <Text>
;
; Der Text wird auf dem Screen ausge-
; gegeben. Der Tastencode wird an DOS
; zurückgegeben. (Bei Funktionstasten
; wird FF zurückgegeben). Er lässt sich
; per ERRORLEVEL abfragen. Das Programm
; ist als COM-Datei zu übersetzen!!
;=====
;
;   .MODEL SMALL
;   .RADIX 16           ; Hexadezimalsystem
;
;   Blank EQU 20       ; Blank
;   Err1  EQU 0FF      ; Error
;
;   .STACK             ; 1 K Stack
;
;   .DATA
;
; Bereich mit den Textkonstanten
;
Crlf  DB 0D, 0A,"$"
;
;
;   .CODE
;
ASK:   CALL NEAR PTR Text ; Textausgabe
      CALL NEAR PTR Key  ; Abfrage der Tastatur
      PUSH AX             ; merke Tastencode
;
; CR,LF ausgeben
;
      MOV AX,SEG Crlf    ; lese Segment Text
      PUSH DS           ; merke DS-Inhalt
      MOV DS,AX         ; setze Segmentadr.
      MOV AH,09         ; INT 21-Stringausgabe
      MOV DX,OFFSET Crlf ; Stringadresse
      INT 21            ; ausgeben
      POP DS            ; restauriere DS
      POP AX            ; restauriere Tastencode
;
; DOS-Exit, Returncode steht bereits in AL
;
      MOV AH,4C         ; INT 21-Exitcode
      INT 21            ; terminiere
;
Text  PROC NEAR
;-----
; Unterprogramm zur Ausgabe des
; Textes aus der Kommandozeile
;-----
; ermittle Lage des PSP über undok.
; Funktion 51 des INT 21
;
      MOV AH,51         ; DOS-Code

```

```

        INT 21          ; Get PSP
        MOV ES,BX      ; ES:= PSP-Adr
;
; prüfe ob Text im PSP vorhanden ist
;
        MOV CL,ES:[80] ; lese Pufferlänge
        CMP CL,0       ; Text vorhanden ?
        JZ  Ready      ; Nein -> Exit
;
; Text ist vorhanden, ausgeben per INT 21, AH = 02
;
        MOV BX,0082    ; Zeiger auf 2. Zeichen
        DEC CL         ; 1 Zeichen weniger
Loop1:  ; Beginn der Ausgabeschleife !!!!
        MOV AH,02      ; INT 21-Code Display Char.
        MOV DL,ES:[BX] ; Zeichen in DL laden
        INT 21         ; CALL DOS-Ausgabe
        INC BX         ; Zeiger nächstes Zchn
        DEC CL         ; Zeichenzahl - 1
        JNZ Loop1     ; Ende ? Nein-> Loop
;
        MOV AH,02      ; INT 21-Code Display
        MOV DL,Blank   ; Blank anhängen
        INT 21         ; und ausgeben
Ready:  RET           ; Ende Unterprogramm
Text   ENDP

;
Key     PROC NEAR
;-----
; Unterprogramm zur Tastaturabfrage
; benutze INT 21, AH = 08 Read Keyboard
; oder:      AH = 01 Read Keyboard & Echo
;-----
; lese 1. Zeichen
        MOV AH,01      ; INT 21-Read Key & Echo
        INT 21         ; Read Code
        CMP AL,0       ; Extended ASCII-Code ?
        JNZ Exit      ; Nein -> Ready
;
; lese 2. Zeichen beim Extended ASCII-Code
;
        MOV AH,08      ; INT 21 Read Keyboard
        INT 21         ; Code aus Puffer lesen
        MOV AL,Err1    ; Fehlercode setzen
Exit:   RET           ; Ende Unterprogramm
Key     ENDP

END Ask

```

Listing 6.4: ASK.ASM

Das Programm benutzt eine Reihe neuer Funktionen, auf die ich kurz eingehen möchte. Zuerst einmal die Technologie der DOS-Aufrufe zur Erledigung der Aufgabe. Zur Ausgabe der Benutzermeldung ist die INT 21-Funktion AH = 02 zu nutzen, die ein Zeichen im Register DL erwartet und dieses Zeichen ausgibt. Dann fragt ASK die

Tastatur ab. Mit der INT 21-Funktion AH = 01H wartet DOS solange, bis ein Zeichen eingegeben wird. Das Zeichen wird bei der Eingabe durch DOS auf den Ausgabebildschirm kopiert. So sieht der Benutzer seine Eingabe. Nun besitzt ein DOS-Rechner Tasten, die nicht ein Byte, sondern zwei Byte (Extended ASCII Code) zurückgeben. Funktionstasten gehören zu dieser Kategorie. Wird eine Taste mit erweitertem Code (Extended ASCII-Code) betätigt, hat das erste gelesene Byte den Wert 00H. Dann muß das Programm den Tastaturpuffer ein weiteres mal lesen um das zweite Zeichen zu entfernen. Hierzu wird die INT 21-Funktion 08H benutzt, die kein Echo der eingegebenen Zeichen zuläßt. Um innerhalb eines Batchprogramms zu erkennen, daß eine Funktionstaste gedrückt wurde, sollte ASK in diesem Fall den Wert FFH an DOS zurückgeben. Damit entfallen die Funktionstasten zur Eingabe. Die INT 21-Funktionen AH = 01H und AH = 08H geben das gelesene Zeichen im Register AL zurück. Wird nun die INT 21-Funktion AH = 4CH aufgerufen, terminiert das Programm ASK. Der Code im Register AL läßt sich dann über ERRORLEVEL abfragen. Mit diesem Wissen sollte sich das Problem eigentlich lösen lassen.

Leider bringt die Verwendung von EXE-Dateien eine Reihe weiterer Schwierigkeiten mit sich. So kann die Lage des PSP nicht mehr so einfach wie bei COM-Dateien ermittelt werden. Vielmehr ist der (undokumentierte) INT 21-Aufruf AH=51H zu nutzen. Dieser gibt im Register BX die Lage des PSP-Segementes zurück.

## 6.12 Die PROC-Directive

Um das Programm ASK transparenter zu gestalten wurden einzelne Funktionen in Unterprogramme verlagert. Die Ausgabe des Textes der Kommandozeile erfolgt in dem Unterprogramm *Text*. Für die Tastaturabfrage ist das Modul *Key* zuständig. Die Prozeduren werden mit den Schlüsselworten:

```
xxx PROC NEAR
....
xxxx ENDP
```

eingeschlossen. Damit erkennt der Assembler, daß er ein RET-NEAR als Befehl einsetzen muß. Beim Aufruf der Prozedur kann TASM auch prüfen, ob der Abstand größer als 64 KByte wird.

Diese Anweisung muß bei TASM am Beginn eines Unterprogrammes (Prozedur) stehen. Dabei sind die Schlüsselworte:

```
Name PROC FAR
Name PROC NEAR
Name PROC
```

zulässig.

Name steht dabei für den Namen der Prozedur, während die Schlüsselworte FAR und NEAR die Aufrufform für den CALL-Befehl festlegen. Sobald ein RET-Befehl auftritt, ersetzt der Assembler diesen durch die betreffende RET- oder RETF-Anweisung. Der Programmierer braucht also nur einen Befehlstyp für RET einzusetzen und der Assembler ergänzt den Befehl. Viele Programmierer ziehen aber vor, den RET-Befehl explizit (notfalls auch als RETF) zu schreiben. Dies ist bei der Wartung älterer Programme hilfreich, da sofort erkennbar wird, ob eine Prozedur mit FAR oder NEAR aufzurufen ist. Wird eine Prozedur mit PROC definiert, muß am Ende des Programmes das Schlüsselwort ENDP stehen. Da das SMALL-Model vereinbart wurde, sind alle Unterprogrammaufrufe als NEAR kodiert.

### 6.12.1 Anmerkungen

Bei einem EXE-Programm läßt sich ein Datensegment mit der DATA-Directive erzeugen. Ein Problem ist die Zuweisung eines korrekten Wertes für das Segmentregister DS. Der TASM bietet hierzu die folgende Möglichkeit:

```
MOV AX,@DATA
MOV DS,AX
```

die die erforderlichen Anweisungen erzeugen.

Um aus dem Quellprogramm eine EXE-Datei zu erzeugen, sind mehrere Schritte erforderlich:

- ◆ Erstellen der Quelldatei
- ◆ Übersetzen in einen OBJ-File
- ◆ Linken des OBJ-Files zu einer EXE-Datei

Die Übersetzung der Quelldatei in eine OBJ-Datei erfolgt mit der Anweisung:

```
TASM ASK.ASM,,,
```

Der Assembler legt seine Fehlermeldungen dann in der Listdatei ASK.LST ab. Bei einer fehlerfreien Übersetzung ist die OBJ-Datei in eine EXE-Datei zu linken:

```
LINK ASK.OBJ
```

Der Linker wird dann die Namen der EXE-, MAP- und LIB-Dateien abfragen. Um diese Abfrage zu umgehen, kann hinter dem OBJ-Namen eine Serie von Kommas folgen:

```
LINK ASK.OBJ,,,,
```

Damit wird LINK signalisiert, daß die Namen der jeweiligen Ausgabedateien aus dem Namen der Eingabedatei (hier ASK) zu extrahieren ist.

Sobald die Datei ASK.EXE vorliegt, läßt sie sich mit einem Debugger testen.

## 6.13 Erstellen einer EXE-Datei aus mehreren OBJ-Files

Nun möchte ich noch einen Schritt weitergehen und ein EXE-Programm aus mehreren Modulen aufbauen. Bei größeren Programmen möchte man nicht alle Unterprogramme in einer Quelldatei stehen haben. Vielmehr wird man einzelne Unterprogramme und Module in getrennten Dateien halten. Dies erleichtert das Editieren und Übersetzen. Sobald die Datei in einen OBJ-File übersetzt wurde, läßt sich dieser mit LINK zu den Hauptprogrammen zubinden. Diese Technik möchte ich kurz vorstellen.

### Programmbeispiel

Unser Beispielprogramm erhält den Namen:

ESC.EXE

und erlaubt die Ausgabe von Zeichen an die Standard-Ausgabeeinheit. Die Zeichen lassen sich als Strings oder Hexzahlen beim Aufruf eingeben. Damit gilt folgende Aufrufsyntax:

ESC <Param1> <Param2> .... <Param n>

Die Parameter <Param x> enthalten die auszugebenden Zeichen als:

- ◆ Hexzahl
- ◆ String

Die Parameter sind durch Kommas oder Leerzeichen zu trennen. Ein String zeichnet sich dadurch aus, daß die Zeichen durch Anführungszeichen eingerahmt werden. Mit ESC lassen sich zum Beispiel sehr komfortabel Steuerzeichen an den Drucker übergeben. Folgende Zeilen enthalten Beispiele für den Aufruf des Programmes. Die Kommentare sind nicht Bestandteil des Aufrufes:

```
ESC 41 42 43           ; erzeuge ABC auf Screen
ESC 0C > PRN:         ; Seitenvorschub auf PRN:
ESC 1B 24 30 > PRN:   ; Steuerzeichen an PRN:
```

```
ESC "Hallo" 0A 0D      ; Hallo auf Screen
ESC "Text" > A.BAT    ; in Datei schreiben
```

In der Vergangenheit hat mir das Programm gute Dienste geleistet. So lässt sich mit einigen Batchbefehlen der Drucker auf beliebige Schriftarten einstellen. Im Anhang ist ein Batchprogramm als Beispiel enthalten. Weitere Hinweise finden sich in /2/.

Das Programm ESC.EXE wird in drei Quelldateien aufgeteilt. Eine Datei enthält ein Unterprogramm zur Wandlung einer Hexzahl in Form eines ASCII-Strings in den entsprechenden Hexadezimalwert. Die Quelldatei besitzt folgendes Format:

```

;=====
; File : HEXASC.ASM (c) G. Born
; Version: 1.0 (TASM)
; Convert ASCII-> HEX
; CALL: ES:DI -> Adresse 1.Ziffer (XX)
;       CX       Länge Parameterstring
; Ret.: CY : 0   o.k.
;       AL       Ergebnis
;       ES:DI -> Adresse nächstes Zeichen
;       CY : 1   Fehler
;=====
;
        .MODEL SMALL
        .RADIX 16           ; Hexadezimalsystem
        PUBLIC HexAsc      ; Label global

Blank EQU 20               ; Leerzeichen
Null  EQU 30

        .CODE
;
HexAsc: MOV AL,ES:[DI]      ; lese ASCII-Ziffer
        CMP AL,'a'         ; Ziffer a - f ?
        JB L1              ; keine Kleinbuchstaben
        SUB AL,Blank       ; in Großbuchstaben
L1:     CMP AL,Null        ; Ziffer 0 - F ?
        JB Error1         ; keine Ziffer -> Error
        CMP AL,'F'        ; Ziffer 0 - F ?
        JA Error1         ; keine Ziffer -> Error
        SUB AL,Null       ; in Hexzahl wandeln
        CMP AL,9          ; Ziffer > 9 ?
        JBE Ok            ; JMP OK
        SUB AL,07         ; korr. Ziffern A..F
        JO Error1         ; keine Ziffer -> Error
Ok:     CLC               ; Clear Carry
        RET               ; Exit
; -> setze Carry
Error1: STC               ; Error-Flag
        RET               ; Exit
;
        END HexAsc

```

Listing 6.5: HEXASC.ASM

Das Unterprogramm erwartet in ES:DI einen Zeiger auf das erste Zeichen der zu konvertierenden Zahl. HEXASC konvertiert immer zwei Ziffern zu einem Byte. Das Ergebnis wird in AL (als Byte) zurückgegeben. Ist das Carry-Flag gelöscht, dann ist der Wert gültig. Bei gesetztem Carry-Flag wurde eine ungültige Ziffer gefunden und das Ergebnis in AL ist undefiniert.

In einer zweiten Quelldatei wurden zusätzliche Hilfsmodule untergebracht:

## SKIP

Aufgabe dieses Moduls ist es, innerhalb des Eingabestrings die Separatorzeichen Blank und Komma zu erkennen. Der Eingabestring wird durch ES:DI adressiert. Erkennt SKIP einen Separator, wird der Lesezeiger solange erhöht, bis kein Separatorzeichen mehr vorliegt oder das Ende des Strings erreicht wurde. Wurde das Ende des Strings erreicht, markiert SKIP dies durch ein gesetztes Carry-Flag. Der Lesezeiger ES:DI zeigt nach dem Aufruf immer auf das nächste Zeichen.

## String

Kommt in der Eingabezeile Text vor:

```
ESC "Hallo" 0D 0A
```

wird *String* benutzt um den Text direkt an die Ausgabeinheit auszugeben. Die Funktion erwartet den Lesezeiger auf den Text in ES:DI. Wird im ersten Zeichen ein " gefunden, gibt das Unterprogramm die folgenden Zeichen aus. Das Unterprogramm terminiert, sobald ein zweites Anführungszeichen auftritt, ohne daß das String-Ende erreicht ist. Wird das Ende der Parameterliste erreicht, ohne daß der String beendet wurde (zweites Anführungszeichen " fehlt), gibt das Modul ein gesetztes Carry-Flag. Nach der Ausgabe des Strings zeigt der Lesezeiger auf das nächste Zeichen hinter dem String.

## Number

Dieses Modul wertet die Parameterzeile aus und liest eine Hexadezimalzahl mit 2 Ziffern ein. Anschließend wird das Unterprogramm HEXASC zur Konvertierung aufgerufen. Der zurückgegebene Wert wird dann an die Ausgabeinheit gesendet. Das Unterprogramm erwartet wieder einen Lesezeiger in den Registern ES:DI. Nach dem Aufruf wird das Zeichen hinter der Hexzahl durch diesen Zeiger adressiert. Ist das Carry-Flag gesetzt, wurde das Ende der Parameterzeile erreicht. Falls ein fehlerhaftes Zeichen erkannt wird, terminiert das Programm über die DOS-Exit-Funktion.

Das Quellprogramm besitzt folgenden Aufbau:

```

;=====
; File : MODULE.ASM (c) G. Born
; Version: 1.0 (TASM)
; File mit den Modulen zur Ausgabe und Be-
; arbeitung der Zeichen.
;=====
;
        .MODEL SMALL
        .RADIX 16          ; Hexadezimalsystem
        EXTRN HexAsc:NEAR ; Externes Modul

        .CODE
;
;=====
; SKIP Separator (Blank, Komma)
;
; Aufgabe: Suche die Separatoren Blank oder
;         Komma und überlese sie.
;
; CALL: ES:DI -> Adresse ParameterString
;       CX      Länge Parameterstring
; Ret.: CY : 0 o.k.
;       ES:DI -> Adresse 1. Zchn. Parameter
;       CY : 1 Ende Parameterliste erreicht
;=====
;
        PUBLIC Skip
;
Skip    PROC NEAR
Loops: CMP CX,0000      ; Ende Parameterliste ?
        JNZ Test1      ; Nein -> JMP Test
        STC            ; markiere Ende mit Carry
        JMP SHORT Exit2 ; JMP Exit2
Test1:  CMP BYTE PTR ES:[DI], ' ' ; Zchn. = Blank ?
        JZ   Skip1     ; Ja -> Skip
        CMP BYTE PTR ES:[DI], ',' ; Zchn. = "," ?
        JNZ Exit1     ; Nein -> Exit1
Skip1:  DEC CX          ; Count - 1
        INC DI         ; Ptr to next Char.
        JMP NEAR PTR Loops ; JMP Loop
Exit1:  CLC            ; Clear Carry
Exit2:  RET
Skip    ENDP
;
;-----
; Display String
;
; Aufgabe: Gebe einen String innerhalb der
;         Parameterliste aus.
;
; CALL: ES:DI -> Adresse "....." String
;       CX      Länge Parameterstring
; Ret.: CY : 0 o.k.
;       ES:DI -> Adresse nach String
;       CY : 1 Ende Parameterliste erreicht
;-----
;
        PUBLIC String
;

```

```

String PROC NEAR
Begin: CMP BYTE PTR ES:[DI],22 ; " gefunden ?
      JNZ Exit3             ; kein String -> EXIT
      INC DI                ; auf nächstes Zeichen
      DEC CX                ; Count - 1
Loop1: CMP CX,0000         ; Ende Parameterliste ?
      JNZ Test2            ; Nein -> JMP Test
      STC                  ; markiere Ende mit Carry
      RET                  ; Exit
Test2: CMP BYTE PTR ES:[DI],22 ; " -> Stringende ?
      JZ Ende              ; Ja -> JMP Ende
Write: MOV DL,ES:[DI]     ; lese Zeichen
      MOV AH,02           ; DOS-Code
      INT 21              ; ausgeben
      DEC CX              ; Count - 1
      INC DI              ; Ptr to next Char.
      JMP NEAR PTR Loop1  ; JMP Loop
Ende:  INC DI              ; auf Stringende
      DEC CX              ; Count - 1
Exit3: CLC                ; ok-> clear Carry
      RET
String ENDP
;
;-----
; Display Number
;
; Aufgabe: Gebe eine HEX-Zahl innerhalb der
;         Parameterliste aus.
;
; CALL: ES:DI -> Adresse 1.Ziffer (XX)
;       CX      Länge Parameterstring
; Ret.: CY : 0 o.k.
;       ES:DI -> Adresse nach Zahl
;       CY : 1 Ende Parameterliste erreicht
;-----
;
      PUBLIC Number
;
Number PROC NEAR

      CALL NEAR PTR HexAsc ; 1. Ziffer konvert.
      JC Error            ; keine Ziffer -> Error
      MOV DL,AL           ; merke Ergebnis
; 2. Ziffer lesen
      INC DI              ; nächstes Zeichen
      DEC CX              ; Zähler - 1
      CMP CX,0000         ; Pufferende ?
      JZ Displ1          ; JMP Display
      CALL NEAR PTR HexAsc ; 2. Ziffer konvert.
      JC Displ1          ; no Ziffer ->JMP Displ1
;
; schiebe 1. Ziffer in High Nibble
;
      PUSH CX             ; schiebe 1. Ziffer
      MOV CL,04           ; in High Nibble
      SHL DL,CL
      POP CX
      OR DL,AL           ; Low Nibble einblenden

```

```

Displ1: MOV AH,02          ; DOS-Code
        INT 21            ; Code in DL ausgeben
        AND CX,CX         ; Ende Parameterliste?
        STC              ; set Carry zur Vorsicht
        JZ Exit4         ; Ende erreicht -> Exit
        INC DI           ; auf nächstes Zeichen
        DEC CX           ; Count - 1
        CLC              ; Clear Carry
Exit4:  RET
;-----
; Ausgabe des Fehlertextes und Exit zu DOS
;-----
Error:  MOV AX,SEG Txt    ; DS: auf Text
        MOV DS,AX
        MOV AH,09        ; DOS-Code
        MOV DX,OFFSET Txt ; Adr. String
        INT 21           ; Ausgabe
        MOV AX,4C01      ; DOS-Code
        INT 21           ; Exit
;
;=====
; Fehlertext
;=====
Txt DB "Fehler in der Parameterliste",0D,0A,"$"
;
Number ENDP
        END

```

Listing 6.6: MODULE.ASM

Damit wird das eigentliche Hauptprogramm sehr kurz. Es muß lediglich die Adresse der DOS-Kommandozeile ermitteln und dann die Parameter auswerten. Zur Auswertung werden dann die Unterprogramme benutzt. Das Programm besitzt folgenden Aufbau:

```

;=====
; File : ESC.ASM (c) G. Born
; Version: 1.0 (TASM)
; Programm zur Ausgabe von Zeichen an die
; Standardausgabeeinheit. Das Programm wird
; z.B. mit: ESC 0D,0A,1B "Hallo"
; von der DOS-Kommandoebene aufgerufen und
; gibt die spezifizierten Codes aus.
;=====
;
        .MODEL SMALL
        .RADIX 16          ; Hexadezimalsystem

Blank EQU 20              ; Blank

        EXTERN HexAsc:NEAR ; externe Module
        EXTERN String:NEAR
        EXTERN Number:NEAR
        EXTERN Skip:NEAR

;=====

```

```

; Definition des Stacksegmentes
;=====
        .STACK 200H

        .CODE
;
;=====
; Hauptprogramm
;=====
;
Start:  MOV AH,51          ; ermittle Adr. PSP
        INT 21           ; "
        MOV ES,BX        ; ES = Adr. PSP !
        MOV CL,ES:[80]  ; lade Pufferlänge
        CMP CL,0        ; Text vorhanden
        JZ  Endx         ; kein Text vorhanden
        XOR CH,CH       ; clear Counter Hbyte
        MOV DI,0081     ; lade Puffer Offset
Loopb:  ;
        AND CX,CX       ; Ende erreicht ?
        JZ  Endx        ; DOS-Exit
        CALL Skip       ; Separatoren überlesen
        JC  Endx        ; DOS-Exit
        CALL String     ; String ausgeben
        JC  Endx        ; DOS-Exit
        CALL Skip       ; Separ. überlesen
        JC  Endx        ; DOS-Exit
        CALL Number     ; Zahl ausgeben
        JC  Endx        ; DOS-Exit
        JMP Loopb       ; next Param.
;
; DOS-Exit, Returncode in AL
;
Endx:  MOV AX,4C00      ; DOS Exitcode
        INT 21
;
        END Start

```

*Listing 6.7: ESC.ASM*

Die einzelnen Quelldateien sind mit den Anweisungen:

```

TASM ESC.ASM,,,,
TASM MODULE.ASM,,,,
TASM HEXASC.ASM,,,,

```

zu übersetzen. Der Macroassembler wird zwar einige Fehlermeldungen bringen. So signalisiert er zum Beispiel, daß in ASK verschiedene Prozeduren (Skip, etc.) aufgerufen werden, die nicht in der Quelldatei vorhanden sind. Dies ist nicht weiter tragisch, daß anschließend die einzelnen OBJ-Files mit LINK kombiniert werden:

```

TLINK ESC+MODULE+HEXASC,,,,

```

Damit werden die OBJ-Files zu einem lauffähigen EXE-Programm kombiniert. Der Linker löst auch die vom Assembler gemeldeten externen Referenzen auf. Sofern der Linker keine Fehlermeldungen mehr bringt kann nun das Programm ESC.EXE mit einem Debugger getestet werden.

## 6.14 Die Directiven für externe Module

In obigen Quellprogramme finden sich einige neue Anweisungen für den Assembler die kurz vorstellen möchte. Die nachfolgenden Directiven beziehen sich auf getrennt zu assemblierende Module und sind für den Linker erforderlich.

### 6.14.1 Die PUBLIC-Directive

Diese Anweisung erlaubt die explizite Auflistung von Symbolen (Variable, Prozeduren, etc.), die durch andere Module adressierbar sein sollen. Der Linker wertet diese Informationen aus den .OBJ-Dateien aus und verknüpft offene Verweise auf diese Symbole mit den entsprechenden Adressen. So kann zu einem Programm eine Prozedur aus einer fremden Objektdatei zugebunden werden. Der Linker setzt dann im CALL-Aufruf die Adresse des Unterprogrammes ein. Weiterhin lassen sich mit der PUBLIC-Directive Variable als global definieren, so daß andere Module darauf zugreifen können. Die Anweisung sollte im Modulkopf stehen und kann folgende Form besitzen:

```
PUBLIC Skip, Number, String
```

Die Definitionen dürfen aber auch direkt vor dem jeweiligen Unterprogramm angegeben werden.

Die Symbole Skip, String und Number lassen sich dann aus anderen Modulen (hier aus ESC) ansprechen, wenn sie dort als EXTRN erklärt werden. Fehlt die PUBLIC-Anweisung, wird der Linker eine Fehlermeldung mit den offenen Referenzen angeben. Zur Erhöhung der Programtransparenz und zur Vermeidung von Seiteneffekten sollten nur die globalen Symbole in der PUBLIC-Anweisungsliste aufgeführt werden.

### 6.14.2 Die EXTRN-Directive

Diese Directive ist das Gegenstück zur PUBLIC-Directive. Soll in einem Programm auf ein Unterprogramm oder eine Variable aus einem anderen .OBJ-File zugegriffen werden, muß dies dem TASM bekannt sein. Mit der EXTRN-Directive nimmt der TASM an, daß das Symbol in einer externen .OBJ-Datei abgelegt wurde und durch den Linker überprüft wird. Die EXTRN-Directive besitzt folgendes Format:

```
EXTRN Name1:Typ, Name2:Typ, ....
```

Als Name ist der entsprechende Symbolname einzutragen. Weiterhin muß der Typ der Referenz in der Deklaration angegeben werden. Hierfür gilt:

```
BYTE oder NEAR  
WORD oder ABS  
DWORD  
QWORD  
FAR
```

Das Gegenstück zu obiger PUBLIC-Definition ist z.B. die Erklärung der externen Referenzen im Hauptprogramm:

```
EXTRN Skip:NEAR, String:NEAR, Number:NEAR
```

Tritt nun eine Referenz auf ein solches Symbol auf (z.B: CALL NEAR PTR Skip), generiert der TASM einen Unterprogrammaufruf ohne die absolute Adresse einzutragen. Die Adresse wird beim LINK-Aufruf dann nachgetragen.

### 6.14.3 Die END-Directive

Diese Directive signalisiert das Ende des Assemblermoduls. Die Anweisung besitzt die Form:

```
END  
END start_adr
```

wobei der Name *start\_adr* als Label für Referenzen dienen darf. TASM behandelt den hinter END angegebenen Namen wie ein Symbol, welches mit EQU definiert wurde. So läßt sich dann die Programmlänge leicht ermitteln. Folgen weitere Anweisungen, werden diese als getrenntes Programm übersetzt. Damit lassen sich in einer Quell-codetei mehrere Module ablegen und mit einem Durchlauf assemblieren.

### 6.14.4 Die GROUP-Directive

Mit dieser Directive wird der Linker angewiesen, alle angegebenen Programmsegmente innerhalb eines 64-KByte-Blocks zu kombinieren. Es gilt dabei die Syntax.

```
Group_name GROUP Seg_name1, Seg_name2, ...
```

Der Group\_name läßt sich dann innerhalb der Module der Gruppe verwenden (z.B: MOV AX,Group\_name). Passen die Module nicht in einen 64-KByte-Block, gibt der Linker eine Fehlermeldung aus. Mit dieser Anweisung lassen sich einzelne Module zu

Gruppen kombinieren und in einem Segment ablegen. Zu Beginn des Segmentes werden die Segmentregister gesetzt und sind für alle Module der Gruppe gültig. Wird die GROUP-Directive verwendet, muß sie allerdings in allen Modulen benutzt werden, da sonst einzelne Module nicht im Block kombiniert werden. Die GROUP-Anweisung wird in den Beispielprogrammen nicht benutzt.

## 6.15 Einbinden von Assemblerprogrammen in Hochsprachen

Abschließend möchte ich noch kurz auf die Einbindung eines Assemblerprogramms in Hochsprachen eingehen. Hierzu ist das Modul wie in obigem Beispiel als Prozedur in einer Quelldatei zu speichern und durch den TASM in eine OBJ-Datei zu übersetzen. Dann kann das Modul per LINK in eine Hochsprachenapplikation eingebunden werden.

### Programmbeispiel

Nachfolgend möchte ich kurz an Hand eines kleinen Beispiels die Einbindung von Assemblerprogrammen in Turbo Pascal- und QuickBasic-Programme zeigen.

Aufgabe ist es eine Prüfsumme nach dem CRC16-Verfahren zu berechnen. Diese Operation wird bei der Datenübertragung zur Fehlerprüfung häufig verwendet. Die Realisierung einer CRC-Berechnung ist in Hochsprachen nicht effizient möglich. Deshalb wird die Routine in Assembler kodiert und als OBJ-File in die Hochsprachenapplikation eingebunden.

Das Programm CRC.ASM übernimmt die Berechnung der CRC-Summe. Die Parameter werden per Stack übergeben. Das Modul ist als OBJ-File zu übersetzen. Der genaue Aufbau der Parameterübergabe und des Programmes ist nachfolgendem Listing zu entnehmen.

```
-----  
; File      : CRC.ASM  
; Version   : 1.1 (TASM)  
; Autor     : (c) G. Born  
; Funktion  :  
;  
;          CRC 16 Generator für den 8086 Prozessor  
;  
; Es wird das Polynom  $y = x^{16} + x^{15} + x^2 + 1$   
; verwendet  
; Aufruf von Hochsprachen mit:  
;  
; CALL CRC (CRCr, Buff, Len)  
;  
; CRCr = Adresse Variable 16 Bit CRC Register
```

```

; Buff = Adresse des Zeichenpuffers
; Len = Wert Zeichenzahl im Puffer
;
; Es sind die Adressen der zwei Parameter CRCr,
; Buff und der Wert von Len über den Stack zu
; übergeben:
;
;
;
;           Stack Anordnung
;   +-----+
;   ! Seg:   CRCr           !
;   +-----+ + 0E
;   ! Ofs:   CRCr           !
;   +-----+ + 0C
;   ! Seg:   Buff           !
;   +-----+ + 0A
;   ! Ofs:   Buff           !
;   +-----+ + 08
;   ! Wert   Len            !
;   +-----+ + 06
;   ! Seg:   Return Adr.!
;   +-----+ + 04
;   ! Ofs:   Return Adr.!
;   +-----+ + 02
;   ! BP   von CRC         !
;   +-----+
;
; Die Procedur ist als FAR aufzurufen. Es wird
; angenommen, daß die Variablen im DS-Segment
; liegen. Diese Konvention entspricht der
; Parameterübergabe in Turbo Pascal und Quick
; Basic. Für die eigentliche CRC-Berechnung
; gilt folgende Registerbelegung:
;
; Register: BX = CRC Register
;           SI = Zeiger in den Datenstrom
;           CX = Zahl der Daten
;           AH = Bit Counter
;           AL = Hilfsaccu
;
;-----
;
;           .RADIX      16
;           .MODEL      LARGE
;
POLY      EQU          001000000000001B ; Polynom
;
;           .CODE
;
CRC       PUBLIC      CRC
          PROC        FAR
;
CRCX:     PUSH        BP           ; save old frame
          MOV         BP,SP       ; set new frame
          PUSH        DS         ; save DS
          MOV         CX,[BP]+06  ; get (Len)
          MOV         SI,[BP]+08  ; get Adr (Buff)
          MOV         DS,[BP]+0A  ; get Seg (Buff)

```

```

        MOV     BX,[BP]+0C    ; get Adr (CRC)
        MOV     DI,BX        ; merke Adr
        MOV     BX,DS:[BX]   ; load CRC value
;
        CLD                 ; Autoincrement
;
LESE:   MOV     AH,08        ; 8 Bit / Zeichen
        LODSB                ; get Zchn & SI+1
        MOV     DL, AL       ; merke Zeichen
;
; CRC Generator
;
CRC1:   XOR     AL,BL        ; BCC-LSB XOR Zchn
        RCR     AL,1        ; Ergeb. Carry Bit
        JNC     NULL        ; Serial Quot.?
;
; Serial Quotient = 1
;
EINS:   RCR     BX,1        ; SHIFT CRC right
        XOR     BX,POLY     ; Übertrag einbl.
        JMP     TESTE       ; weitere Bits
;
; Serial Quotient = 0
;
NULL:   RCR     BX,1        ; Shift CRC right
;
TESTE:  MOV     AL,DL       ; Lese Zeichen
        RCR     AL,1        ; Zchn 1 Bit right
        MOV     DL,AL       ; merke Rest zchn
        DEC     AH          ; 8 Bit fertig ?
        JNZ     CRC1        ; Zchn ready ?
        LOOP   LESE         ; String ready?
;
; Ende -> Das CRC Ergebnis steht im Register BX
; -> schreibe in Ergebnisvariable zurück
;
        MOV     DS:[DI],BX  ; restore CRC
;
        POP     DS          ; restore Seg-reg.
        POP     BP          ; rest. old frame
        RETF    0A         ; POP 10 Bytes

CRC     ENDP
;
        END

```

Listing 6.8: CRC.ASM

Die Einbindung des OBJ-Moduls in Turbo Pascal 4.0-7.0 ist dann recht einfach. Wichtig ist, daß die Prozedur als FAR aufgerufen wird und die Übergabeparameter korrekt definiert wurden. Das nachfolgendes Listing zeigt beispielhaft wie dies realisiert werden kann.

```
{ ***** }
```

```

File      : DEMO.PAS
Vers.    : 1.1
Autor    : G. Born
Files    : ---
Progr. Spr.: Turbo Pascal 4.0 (und höher)
Betr. Sys. : DOS ab 2.1
Funktion: Das Programm dient zur Demonstration
des Aufrufes von Assemblerprogrammen aus Turbo
Pascal. Es wird das Programm CRC.OBJ einge-
bunden. Die zu übertragenden Zeichen stehen als
Bytes im Feld buff[].
*****}

TYPE Buffer = Array [1 .. 255] OF Byte;
VAR crc_res : Word;           { CRC Register }
    buff : Buffer;           { Datenpuffer }

{*****          Hilfsroutinen          *****}

{*
  Hier wird die OBJ-Datei eingebunden
  und die Prozedur definiert
*}
{$L CRC.OBJ}                 { OBJ. File      }
{$F+}                        { FAR Modell ! }
procedure CRC (var crc_res : word; var buff : Buffer; len :
integer);
external;
{$F-}

procedure Write_hex (value, len : integer);
{
  Ausgabe eines Wertes als Zahl auf dem Bildschirm.
  Durch Len wird festgelegt, ob ein Byte (Len = 1)
  oder Wort (Len = 2) ausgegeben werden soll.
}
const Hexzif : array [0..15] of char = '0123456789ABCDEF';
    Byte_len = 1;
    Word_len = 2;

TYPE zahl = 1..2;
VAR temp : integer;
    carry : zahl;
    i : zahl;
begin
  if len = Word_len then
  begin
    temp := swap (value) and $0FF;           { high byte holen }
    write (Hexzif[temp div 16]:1,Hexzif[temp mod 16]:1);
    end;
    temp := value and $0FF;                 { low byte holen }
    write (Hexzif[temp div 16]:1,Hexzif[temp mod 16]:1);
end; { Write_hex }

{**** Hauptprogramm ****}

begin
  crc_res := 0;                             { clear CRC - Register }

```

```

buff[1] := $55;           { Testcode setzen }
buff[2] := $88;
buff[3] := $CC;

writeln ('CRC - Demo (c) Born G. ');
writeln;
writeln ('CRC-Berechnung per Polynomdivision');
writeln;

CRC (crc_res, buff, 3);   { Aufruf CRC Routine 1 }
write ('Die CRC - Summe ist : ');
write_hex (crc_res,2);   { Hexzahl ausgeben }
writeln;
end.                       { Ende }

```

*Listing 6.9: DEMO.PAS*

Anschließend läßt sich das Programm als EXE-File mit der Eingabe:

DEMO

aufrufen.

Um das Programm CRC.OBJ in QuickBasic 4.x einzubinden, geht man analog vor. Das nachfolgende Listing in QuickBasic demonstriert die Einbindung des OBJ-Moduls.

```

'|*****
'| File      : DEMO.BAS
'| Vers.    : 1.1
'| Autor    : G. Born
'| Files    : ---
'| Progr. Spr.: Quick Basic 4.0 / 4.5
'| Betr. Sys.: DOS 2.1 - 4.01
'| Funktion: Das Programm dient zur Demonstration der Ein-
'|           bindung von Assemblermodulen in QuickBasic.
'|           Es wird das Modul CRC.OBJ benutzt, Die Parameter
'|           stehen als Bytes im Feld buff%(), oder im String
'|           buff$. Aus diesen Zeichen wird dann die CRC16-
'|           Summe mittels der Proedur CRC (File CRC.OBJ)
'|           berechnet. Compiler und Linker sind mit folgenden
'|           Parametern aufzurufen:
'|
'|           BC DEMO.BAS
'|           LINK DEMO.OBJ,CRCA.OBJ
'|
'|*****

DIM buff%(255)           '! Integer Puffer

crcres% = 0              '! clear CRC-Register
'|
'| setze Zeichen in Puffer, beachte aber, daß es kein Byte
'| Datum gibt, d.h. zwei Bytes sind in einer Integer Variablen
'| zu speichern !!!

```

```
'!  
buff%(0) = &H8855           '! setze Zeichen in  
buff%(1) = &HCC             '! INTEGER Puffer  
'!  
'! Setze Zeichen alternativ in den Stingpuffer  
'!  
buff$ = CHR$(&H55)+CHR$(&H88)+CHR$(&HCC) '! Testcode setzen  
  
PRINT "CRC-Demo Programm in Basic (c) Born G."  
PRINT  
PRINT "CRC-Berechnung per Polynomdivision"  
PRINT  
'!  
'! Berechne CRC aus Integer Puffer  
'!  
CALL CRC (SEG crcres%, SEG buff%(0), BYVAL 3) '! Aufruf CRC  
Routine 1  
  
PRINT "Die CRC - Summe ist : ";HEX$(crcres%) '! Hexzahl ausgeben  
  
'!  
'! Berechne CRC aus String Puffer  
'!  
crcres% = 0                 '! clear CRC-Register  
FOR i% = 1 TO LEN(buff$)   '! separiere Zeichen  
    tmp% = ASC(MID$(buff$,i%,1)) '! in Integer wandeln  
    CALL CRC (SEG crcres%, SEG tmp%, BYVAL 1) '! CRC Routine 1  
NEXT i%  
  
PRINT  
'! Hexzahl ausgeben  
PRINT "Die CRC - Summe ist : ";HEX$(crcres%)  
END  
'!**** Ende ****
```

*Listing 6.10: DEMO.BAS*

Die Einbindung in andere Programmiersprachen kann analog erfolgen. Gegebenenfalls sind die Compilerhandbücher als Referenz zu benutzen.

Damit möchte ich die Ausführungen zu TASM beenden. Sicherlich konnten nicht alle Aspekte des Produkte vorgestellt werden. Hier ist die entsprechende Originaldokumentation des Herstellers zu konsultieren. Für einen ersten Einstieg sollte der vorliegende Text aber genügen.

---

# 7 DEBUG als Assembler und Werkzeug in DOS.

Zum Lieferumfang des DOS-Betriebssystems gehört auch das Programm DEBUG. Dieses Programm bietet zahlreiche Hilfsfunktionen, die unter anderem auch die Erstellung von kleineren COM-Programmen aus Assembleranweisungen erlauben. Das Produkt besitzt zwar einige Restriktionen und Unbequemlichkeiten was die Kommandoingabe betrifft. Wer aber über keinen Assembler verfügt, kann mit DEBUG alle Übungen aus Kapitel 2 durchführen und erhält eine Reihe interessanter Utilities.

## 7.1 Die Funktion des Debuggers

Die Hauptfunktion des Debuggers besteht in der kontrollierten Ausführung des zu testenden Programmes. Hierzu gehört, daß das Programm an definierten Stellen angehalten, geändert und wieder gestartet werden kann. Für die Assemblerprogrammierung ist es weiterhin nützlich, wenn sich die Speicherinhalte anzeigen und modifizieren lassen. Funktionen zur Übersetzung von Assembleranweisungen in Maschinensprache und die Möglichkeit zur Rückübersetzung sind ebenfalls recht interessant. Alle diese Funktionen werden von dem Programm DEBUG geboten.

### 7.1.1 Der Programmstart

DEBUG besitzt folgende Aufrufsyntax:

```
DEBUG [File] [Parameter]
```

Die in Klammern stehenden Felder [File] und [Parameter] sind optional und können bei der Eingabe entfallen. In [File] kann bereits beim Aufruf der Name des zu testenden Programmes angegeben werden. DEBUG lädt dann direkt den Programmcode in den Speicher, ohne ihn jedoch auszuführen. Falls das Programm nach dem Aufruf bestimmte Parameter aus der Kommandozeile erwartet, lassen sich diese im Feld [Parameter] eintragen. Nähere Ausführungen zu den beiden Aspekten finden sich bei der Beschreibung des NAME-Befehls. Nachfolgend sind einige gültige Aufrufe für den Debugger aufgeführt:

```
DEBUG  
DEBUG ASK.COM  
DEBUG ASK.COM Hallo
```

Nach dem Aufruf des Debuggers aus der DOS-Ebene erscheint der PROMPT - auf dem Bildschirm.

```
C>DEBUG
```

```
-
```

Damit zeigt DEBUG die Betriebsbereitschaft an und erwartet ein Kommando. Eine Zusammenstellung der möglichen Kommandos findet sich in der folgenden Tabelle.

Befehl	Aufrufsyntax
Assemble	A [adresse]
Compare	C adresse range adresse
Dump	D [adresse][länge]
Enter	E [adresse] [bitmuster]
Fill	F [adresse] [länge] [Bitmuster]
Go	G [=adresse] [breakpoint 1 .. 10]
Hexarithm.	H wert1 wert2
Input	I portadresse
Load	L [adresse] [drive sektor sektor]
Move	M [adresse] [länge] [adresse]
Name	N filename [filename ..]
Befehl	Aufrufsyntax
Output	O adresse byte
Proceed	P [adresse] [wert]
Quit	Q
Register	R [registername]
Search	S [adresse] [länge] [wert]
Trace	T [=adresse] [wert]
Unassemble	U [adresse] [länge]
Write	W [adresse] [drive sektor sektor]
XA(llocate)	XA [count]
XD(ealloc.)	XD [handle]
XM(ap EMS)	XM [lpage] [ppage] [handle]
XS	XS

*Tabelle 7.1: DEBUG-Befehle*

Nun noch einige Hinweise auf die Notation: Sofern im Text nichts anderes spezifiziert ist, sind alle Eingaben für DEBUG mit der RETURN-Taste abzuschließen.

DEBUG läßt sich im Kommandomode (-) durch die Eingabe:

```
-Q
```

beenden. Anschließend erscheint der DOS-PROMPT wieder. Im Speicher geladene Programme werden nicht automatisch auf Diskette gesichert. Falls in den nach-

folgenden Beispielen der Bindestrich vor dem Kommando erscheint, handelt es sich um die Ausgabe des Debuggers. Der PROMPT darf natürlich beim Abtippen der Beispiele nicht mit eingegeben werden! Bei den Kommandos dürfen wahlweise Groß- und Kleinbuchstaben verwendet werden, da DEBUG alle Zeichen in Großbuchstaben konvertiert. Ein laufendes Kommando läßt sich durch gleichzeitiges drücken der Tasten <Ctrl>+<C> abbrechen. Sollte dies nicht möglich sein, hilft nur noch ein Systemreset mittels der Tasten <Alt>+<Ctrl>+ <Del>, oder mittels des Netzschalters. Die Bildschirmausgabe läßt sich mit den Tasten <Ctrl>+<S> unterbrechen. Eine Freigabe kann dann durch jede andere beliebige Taste erfolgen. Mit den Tasten <Shift>+<PrtSc> läßt sich der aktuelle Bildschirminhalt auf dem Drucker protokollieren. Mit <Ctrl>+<P> kann die Protokollierung aller Bildschirmausgaben auf dem Drucker ein- und wieder ausgeschaltet werden. Weiterhin lassen sich die DOS I/O-Umleitungen mit DEBUG nutzen. Weitere Hinweise finden sich bei der Beschreibung des ASSEMBLE-Kommandos.

Doch nun möchte ich auf die ersten DEBUG-Befehle eingehen.

## 7.1.2 Der DUMP-Befehl

Der DUMP-Befehl stellt einen Speicherbereich als Folge von Hexzahlen auf dem Bildschirm dar. Es gilt dabei folgende Aufrufsyntax:

D [Seg:[Offs]] [Range]

Die in Klammern stehenden Parameter sind optional und müssen nicht angegeben werden. Das Feld *Seg* bezeichnet den Wert des Segments, während *Offs* dem Offsetanteil der Adresse entspricht. *Range* ist ein weiterer Parameter, der die Zahl der zu bearbeitenden Bytes angibt. Die DUMP-Ausgabe besteht im Standardformat aus 8 Zeilen:

```
-D
0F66:0100  6A 0E 8B 7E EA 8B 56 F2-89 15 8B 7E EA 8B 56 F0  j...~j.Vr...~j.Vp
0F66:0110  89 55 02 8B 7E EA 8B 56-EC 89 55 04 EB 57 8B 7E  .U...~j.Vl.U.kW.~
0F66:0120  EA 8B 55 02 83 C2 01 89-56 E8 C4 5E F4 26 8B 17  j.U..B...VhD^t&..
0F66:0130  32 F6 89 56 E6 8B 56 E8-3B 56 E6 7F 27 89 56 F0  2v.Vf.Vh;Vf.'Vp
0F66:0140  C4 5E F4 03 5E F0 26 8B-17 8B 4E F0 83 C1 FF C4  D^t.^p&...Np.A.D
0F66:0150  5E F4 03 D9 26 88 17 8B-56 F0 42 89 56 F0 4A 3B  ^t.Y&...VpB.VpJ;
0F66:0160  56 E6 75 DC C4 5E F4 26-8B 17 32 F6 83 C2 FF C4  Vfu\D^t&...2v.B.D
0F66:0170  5E F4 26 88 17 C6 06 FE-E1 01 8B 7E EA FF 35 FF  ^t&..F.~a...~j.5.
```

Bild 7.1: Ausgabe des DUMP-Befehls

Die ersten 9 Zeichen einer Zeile geben die Speicheradresse des folgenden Bytes an. Dieses Byte, sowie weitere 15 Werte sind durch zwei Leerzeichen von der Adreßanzeige getrennt. Der Übergang zwischen dem achten und dem neunten Byte wird durch einen Bindestrich markiert. Den Abschluß einer Zeile bildet die ASCII-Darstellung der Werte. Jedem Byte wird ein entsprechender ASCII-Code zugeordnet. Ist ein Zeichen nicht darstellbar, z.B. das Zeichen 0A, wird es durch einen Punkt ersetzt. Durch eine weitere Eingabe des Zeichens:

-D

werden die nächsten acht Zeilen ausgegeben, d.h. DUMP merkt sich die zuletzt ausgegebene Adresse. Bei der Eingabe können an Stelle des Segmentwertes auch die Register CS, DS, ES oder SS stehen. Wird die Segmentadresse weggelassen, z.B.:

-D 0100

übernimmt DUMP den Inhalt des Datensegmentregisters und interpretiert den Wert 0100 als Offset. Soll ein anderes Segmentregister benutzt werden, ist dies beim Aufruf anzugeben:

-D CS:0100

-D SS:0100

-D ES:0100

Die obigen Aufrufe benutzen das Codesegment (CS), das Stacksegment (SS) und das Extrasegment (ES) als Segmentadresse.

Fehlt die Adreßangaben komplett, benutzt DEBUG das Datensegment und den zuletzt eingestellten Offset zur Adressberechnung. Nach dem Start von DEBUG wird der Offset auf den Wert 100 gesetzt, da dies der Adresse des ersten ausführbaren Programmschritts entspricht.

Die letzte Bemerkung gilt der Option [Range], mit der sich die Zahl der auszugebenden Bytes definieren läßt. Dies kann einmal mittels einer Start- und Endadresse erfolgen. Mit:

-D DS:0 0F

werden z.B. 16 Byte des Datensegments ab dem Offset 0000 bis zur Adresse 000F ausgegeben. Alternativ läßt sich der Befehl auch mit einer Längenangabe formulieren:

-D DS:0 L 10

Dabei gibt der Buchstabe L an, daß der folgende Parameter nicht als Adresse, sondern als Länge zu interpretieren ist.

### 7.1.3 Die ENTER-Funktion

Um Speicherbereiche zu ändern, bietet DEBUG die ENTER-Funktion. Dieser Befehl besitzt folgende Aufrufsyntax:

E [Seg:[Offs]] [Bitmuster]

Alle Adreßeingaben erfolgen in der bereits beim DUMP-Befehl beschriebenen Notation.

Geben Sie bitte folgende Kommandos unter DEBUG ein:

```
-E CS:0100
```

Auf dem Bildschirm erscheint dann zum Beispiel folgender Text:

```
-E 4770:0100  
4770:0100 41.
```

ENTER zeigt nach dem Aufruf das erste Byte der angegebene Adresse (A = 41H). Der Punkt signalisiert, daß ENTER auf eine Eingabe wartet. Die Segmentadresse und der erste angezeigte Wert (hier 41H) wird von System zu System, je nach der Speicherbelegung, schwanken. Geben Sie nun bitte folgende Werte ein:

```
40 blank 8F blank blank F0 <RETURN>
```

<RETURN> steht dabei für die RETURN-Taste. Mit *blank* wird hier die Leertaste bezeichnet. In der Eingabezeile sollte in etwa folgendes Bild erscheinen:

```
-E CS:0100  
4770:0100 41.40 07.8F 3E. 07.F0  
-
```

Die Eingaben lassen sich mit der Anweisung:

```
-D CS:100
```

kontrollieren. In den angewählten Bytes sind neue Werte eingetragen. ENTER erlaubt im Abfragemodus verschiedene Eingaben. Nach der Anzeige des Speicherinhaltes gibt ENTER einen Punkt als PROMPT aus und wartet auf eine Eingabe. Hierbei sind folgende Alternativen möglich:

- ◆ Abbruch des Befehls mit der RETURN-Taste. Dann erscheint der DEBUG-PROMPT wieder. Der Wert der zuletzt angezeigten Zelle bleibt erhalten.
- ◆ Die Leertaste schaltet zur folgenden Adresse weiter. Ohne weitere Eingaben bleibt der Wert der zuletzt angezeigten Zelle unverändert.
- ◆ Eingegebene Hexziffern im Bereich zwischen [0 .. FF] werden in den Speicher übertragen und überschreiben die zuletzt angezeigte Zelle. Die Eingabe ist durch ein Leerzeichen abzuschließen.
- ◆ Durch Eingabe des Minuszeichens - wird die vorhergehende Adresse und deren Inhalt in einer neuen Zeile angezeigt.

Sobald mehr als 8 Byte angezeigt wurden, oder nach der Eingabe eines Minuszeichens, beginnt ENTER mit einer neuen Zeile. Mit dem Minuszeichen können also Speicherzellen in absteigender Reihenfolge modifiziert werden.

```
-E
B000:00A4 64.-
B000:00A3 F0.-
B000:00A2 3E.
-
```

Beachten Sie aber, daß immer nur der Offset der Adresse dekrementiert wird, d.h. ENTER arbeitet nur innerhalb eines 64 KByte Bereichs. Sollen gleichzeitig mehrere Bytes modifiziert werden, ohne daß der alte Wert angezeigt wird, kann dies z.B. mit folgender Eingabe erfolgen:

```
-E DS:0200 "Dies ist ein Text"
-
```

Die Bytes "Dies ist.." werden ab der Adresse DS:0200 in den Speicher geschrieben. Dieses Eingabeformat ist vor allem dann nützlich, wenn Texte als ASCII-Zeichen in der Form ".." eingegeben werden müssen. Dann kann die umständliche Umrechnung der Zeichen in Hexwerte entfallen. Alle ASCII-Zeichen sind durch Hochkommas oder Anführungsstriche einzuschließen.

### 7.1.4 Der FILL-Befehl

FILL füllt größere Speicherbereiche mit einem bestimmten Bytemuster. Das Kommando besitzt folgende Aufrufsyntax:

```
F Seg:Offs Lxx Bytemuster
```

Die Adresse spezifiziert den Bereich, ab dem das angegebene Bytemuster abgespeichert wird. Die Eingabe muß gemäß den bereits besprochenen Konventionen erfolgen. Die Länge des zu füllenden Bereiches ist auf jeden Fall im Parameter *Lxx* anzugeben. Die Zeichen *xx* stehen dabei für eine maximal 16 Bit breite Zahl (0-FFFFH). Weiterhin sind im Feld *Bytemuster* die Werte für das zu speichernde Muster anzugeben. Ist der zu füllende Bereich größer als das angegebene Füllmuster, dann wiederholt FILL die Operation so oft, bis der spezifizierte Bereich gefüllt ist. Sollen z.B. 16 Byte ab Adresse CS:100 mit dem Wert FF belegt werden, lautet das Kommando:

```
-F CS:100 L 0F FF
-
```

Das Bytemuster (FF) enthält weniger Bytes als in *Lxx* spezifiziert, deshalb wird die Operation so lange wiederholt, bis alle Bytes überschrieben wurden.

### 7.1.5 Die MOVE-Funktion

MOVE erlaubt ganze Bereiche innerhalb des Speichers zu verschieben. Das Aufrufformat für den Befehl lautet:

M Startadr Endadr Zieladr

Die Adressen sind wie gewohnt in der Segment:Offset-Notation anzugeben. Mit Startadresse wird der Beginn des zu verschiebenden Bereiches spezifiziert. Im Parameter *Endadr* ist die Adresse des letzten zu verschiebenden Bytes definiert. Steht hier der Text:

M XXXX L YYYY ZZZZ

wird der Wert YYYY nicht als Endadresse sondern als Längenangabe interpretiert. Der Parameter *Zieladresse* bestimmt, wohin die Bytes verschoben werden. Überlappende Bereiche werden so kopiert, daß keine Informationen verloren gehen. Nur bei der Start- und Zieladresse darf ein Segment angegeben werden. Dies bedeutet, daß sich maximal der Inhalt eines 64 KByte großen Segments in einem Stück verschieben läßt. Ohne Angabe der Segmentadresse wird das DS-Register benutzt. Die Segmentangabe der Startadresse gilt auch für die Endadresse. Die Differenz zwischen Startadresse und Endadresse gibt die Zahl der zu verschiebenden Bytes an. Geben Sie bitte folgende Zeichen ein:

-M CS:100 20 CS:200

-

Dann wird ein Bereich von 32 Byte im Codesegment zwischen 100 und 200 verschoben. Die Eingabe:

-M CS:0 100 DS:0

-

kopiert z.B. 256 Byte des Codebereichs in das Datensegment. Die Quelldaten bleiben dabei im jeweiligen Adressbereich erhalten.

### 7.1.6 Die INPUT-/OUTPUT-Befehle

Diese Befehle erlauben es, beliebige Ein- / Ausgabeadressen der 8086/8088 Prozessoren anzusprechen. Dabei gilt folgende Aufrufsyntax:

O Port Byte

I Port

Mit dem Kommando:

### O Port Byte

kann ein Bytewert [0..FF] in den Ausgabeport geschrieben werden. Der Adressbereich eines Ports liegt im Bereich zwischen 0 - 64 KByte. Der Befehl:

```
-O 2F8 FF  
-
```

schreibt den Wert FF in den Port mit der Adresse 2F8. Mit dem Befehl:

### I Port

läßt sich der Inhalt der Portadresse [Port] einlesen. Die Eingabe:

```
-I 2F8  
6B  
-
```

liest das Port mit der Adresse 2F8H aus und gibt dann den Wert (z.B. 6B) zurück.

## 7.1.7 Die HEX-Funktion

Das Kommando gibt Hilfestellung bei der Berechnung der Summe und der Differenz zweier Hexdezimalzahlen. Das Kommando besitzt das Format:

```
H Zahl1 Zahl2  
Summe Differenz
```

Mit *Zahl1* und *Zahl2* sind zwei maximal 16 Bit große Hexadezimalzahlen einzugeben, deren Summe und Differenz in der folgenden Zeile angezeigt wird. Versuchen Sie folgende Eingabe:

```
-H 0F 02  
11 0D  
-
```

Die Summe beider Zahlen ist 11H, während ihre Differenz 0DH beträgt.

### 7.1.8 Die COMPARE-Funktion

Mit Hilfe dieser Funktion lassen sich zwei Speicherbereiche auf übereinstimmende Speicherinhalte vergleichen. Die Aufrufsyntax lautet:

```
C [Seg]:Offs Len [Seg]:Offs
```

Die Adressen spezifizieren den Beginn der zwei zu vergleichenden Bereiche. Ohne Segmentangaben wird das DS-Register zur Adressberechnung benutzt. *Len* gibt an, wieviele Bytes zu vergleichen sind. Werden ungleiche Bytes gefunden, erfolgt eine Anzeige in der Form:

```
Adr1 Byte1 Byte2 Adr2
```

Die Anzeige gibt also die Lage und die Werte der unterschiedlichen Speicherzellen an. Ein Eingabebeispiel könnte folgendermaßen aussehen:

```
-C CS:0 L3 DS:0  
0000:0000 FF FE 0100:0000  
0000:0001 FE CD 0100:0001  
0000:0002 CD 55 0100:0002  
-
```

Hier wird ein Ausschnitt des Codebereiches mit dem Datenbereich verglichen. In obigem Fall sind alle Bytes unterschiedlich. *L3* gibt an, daß 3 Byte zu vergleichen sind. Wird keine Abweichung gefunden, meldet sich DEBUG nach dem Vergleich nur mit dem PROMPT "-" zurück.

### 7.1.9 Die SEARCH-Funktion

Um bestimmte Werte im Speicherbereich aufzufinden kann der SEARCH-Befehl benutzt werden. Ihr Eingabeformat lautet:

```
S [Seg]:Offs Len Bytemuster
```

Ohne Angabe der Segmentadresse erfolgt die Suche im Datensegment (DS). *Len* gibt die Zahl der zu durchsuchenden Bytes an. Wird in diesem Bereich das Bytemuster gefunden, zeigt SEARCH anschließend die Anfangsadresse an. Falls das Muster mehrfach im Suchbereich auftritt, werden auch die entsprechenden Adressen angezeigt. Erscheint nur der PROMPT "-" nach Aufruf des Befehles, dann wurde das Muster nicht im angegebenen Bereich gefunden. Eine Eingabe könnte folgendermaßen aussehen:

```
-S CS:100 200 55
1000:0101
1000:01FF
-
```

In diesem Fall taucht das Byte 55 zweimal im Codesegment im Bereich 100 .. 300 auf. Es besteht auch die Möglichkeit, nach ganzen Zeichenketten zu suchen:

```
-S DS:100 L100 "Test"
```

Hier wird ein Bereich von 256 Byte nach dem Muster "Test" abgesucht.

### 7.1.10 Der REGISTER-Befehl

Eine weitere Funktion ermöglicht es, den Inhalt der Prozessorregister anzuzeigen und zu verändern. Dies erfolgt mittels einer Eingabe im Format:

```
R [Register]
```

Wird kein Register angegeben, erscheint die folgende Anzeige.

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1007 ES=1007 SS=1007 CS=1007 IP=0100 NV UP EI PL NZ NA PO NC
1007:0100 0000 ADD [BX+SI],AL DS:0000=CD
-
```

Die erste Zeile enthält die Anzeige der Arbeitsregister, während in der zweiten Zeile die Segmentregister, der Instruktionszähler (IP), sowie der Zustand der Flags dargestellt werden. In der dritten Zeile wird der aktuell im Codesegment per Instruktionszähler angesprochene Befehl decodiert. Findet ein Zugriff auf Daten statt, werden Adresse und Inhalt des entsprechenden Datums angezeigt. Soll nur der Inhalt eines Registers angezeigt oder verändert werden, erfolgt dies durch die Eingabe des Registernamens. Nachfolgendes Beispiel setzt das BX-Register.

```
-R BX
BX 0000
:3456
-
```

Nach Selektion des BX-Registers in der Eingabezeile wird dessen Inhalt in der zweiten Zeile dargestellt. In der dritten Zeile erscheint die Abfrage des neuen Wertes. Dies ist am PROMPT in Form des Doppelpunktes erkennbar (:). Durch die RETURN-Taste wird die Eingabe abgebrochen ohne den Registerwert zu verändern. In obigem Beispiel wird der Registerinhalt auf den Wert 3456 gesetzt und durch die RETURN-Taste abgeschlossen.

Der Zustand der Flags kann durch:

```
-R F
OV DN EI NG ZR AC PE CY-
```

abgerufen werden. Der Bindestrich am Ende der zweiten Zeile signalisiert, daß DEBUG auf eine Eingabe wartet. Mit RETURN wird die Anzeige ohne Änderungen verlassen. Sollen bestimmte Flags modifiziert werden, ist der jeweils komplementäre Wert in der aktuellen Anzeigezeile einzugeben:

```
-R F
OV DN EI NG ZR AC PE CY-NCNZ
-
```

Mit NC wird das Carry-Flag gelöscht, während NZ das Zero-Flag beeinflusst. Die folgende Tabelle beinhaltet die Bezeichnung der Flags und deren Zustände.

Flag	Set	Clear
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxillary Carry	AC	NA
Parity	PE	PO
Carry	CY	NC

*Tabelle 7.2: Bezeichnung der Flags*

Mit Hilfe der REGISTER-Funktion lassen sich insbesondere die Segmentregister für andere DEBUG-Befehle voreinstellen. Weiterhin wird der Befehl im Zusammenhang mit der weiter unten vorgestellten WRITE-Funktion benötigt.

Beim Start von DEBUG werden die Segmentregister auf den untersten freien Speicherbereich gesetzt. Das Register IP enthält den Wert 100. Wurde ein Programm durch DEBUG geladen, steht die Zahl der Bytes in den Registern BX:CX.

### 7.1.11 Die File I/O-Befehle

DEBUG enthält einige Funktionen zum Einlesen und Speichern von Dateien und Platten-/Diskettensegmenten. Dies Funktionen werden nachfolgend beschrieben.

### 7.1.12 Der NAME-Befehl

Um ein Programm zu Speichern oder zu Laden muß zuerst ein Dateiname angegeben werden. Hierfür existiert die NAME-Funktion mit dem Format:

```
N [file1] [file2] [Param]
```

DEBUG speichert Dateinamen zum Laden der Dateien ab. Im Feld definierte Parameter werden im Programm-Segment-Prefix (PSP) des Programmes ab CS:81 gespeichert. Die LOAD- und WRITE-Kommandos greifen auf diese Informationen zu. Der erste Name wird als Filename der zu ladenden oder zu beschreibenden Datei interpretiert. Mit der Anweisung:

```
N C:COMMAND.COM
```

wird die nächste WRITE- oder READ-Anweisung auf die Datei COMMAND.COM zugreifen. Wichtig ist, daß beim Aufruf die Extension des Dateinamens mit angegeben wird.

### 7.1.13 Der WRITE-Befehl

Nach der Definition des Dateinamens per NAME-Befehl ist DEBUG bereit ein WRITE-Kommando mit dem Format:

```
W [adress [drive sector sector]]
```

auszuführen. Ab der angegebenen Adresse werden nun die Bytes in einer Datei abgelegt. Es gelten die bereits besprochenen Konventionen bezüglich der Segmentangabe. Standardmäßig wird das CS-Register (CS:100) benutzt. Durch Angabe des Laufwerkes A=0, B=1, etc. und der Disksektoren können auch absolute Sektoren einer Disk beschrieben werden. Da dies aber eine sehr unsichere Sache ist, bei Angabe von falschen Werten wird der Inhalt der Diskette zerstört, sollte diese Möglichkeit nur mit großer Vorsicht benutzt werden. WRITE setzt voraus, daß vorher mit NAME ein korrekter Dateiname spezifiziert wurde. Nun stellt sich noch die Frage, wieviele Byte abzuspeichern sind. Diese Information muß mit dem R-Befehl in die BX- und CX-Register (BX:CX) geschrieben werden. Wurde keine Startadresse angegeben, übernimmt WRITE automatisch alle Bytes ab CS:100. Bei der Eingabe des Dateinamens dürfen die Attribute .HEX und .EXE nicht verwendet werden, da DEBUG diese Dateiformate nicht unterstützt. Ein COM-Programm wird mit folgenden Eingaben:

```
-N A:ASK.COM  
-R CX  
CX 0000  
:40  
-R BX
```

```
BX 0000
:
-W 100
-
```

in der Datei ASK.COM abgespeichert. Obige Sequenz speichert dabei 40H Byte. Wird kein NAME-Kommando ausgeführt, benutzt DEBUG die zufällig im File-Control-Block stehenden Parameter. Dies kann z.B. der beim Aufruf des Debuggers angegebene Filename des zu ladenden Programmes sein. Um Fehler zu vermeiden, sollte vor jedem WRITE-Kommando der Dateiname mit NAME definiert werden.

### 7.1.14 Der LOAD-Befehl

Das Laden eines Programmes erfolgt mit der LOAD-Funktion:

```
L [adress] [drive sector sector]
```

Das Format entspricht der WRITE-Anweisung und erlaubt auch absolute Disksektoren zu lesen. Bei Dateizugriffen wird die NAME-Funktion zur Eingabe des Dateinamens benutzt. Mit der Eingabe:

```
-N A:ASK.COM
-L
-
```

wird der in der Datei ASK.COM abgespeicherte Code geladen. Da keine Adresse angegeben wurde, legt LOAD den Code ab CS:100 ab. Die Zahl der Bytes wird an Hand der Dateinformationen automatisch bestimmt. Diese Information findet sich nach dem Ladevorgang in den BX- und CX-Registern (BX: CX). Wurde beim Dateinamen die Extension .EXE oder .HEX spezifiziert, kann die Adresse nicht mehr angegeben werden, da die Adreßeinstellung aus der jeweiligen Datei entnommen wird. Ausführbare Programme werden durch DOS standardmäßig auf die Adresse CS:100 gesetzt. Lediglich bei HEX-Files addiert DEBUG die angegebene Adresse als Offset zu der Ladeadresse in der Datei.

Neben dem expliziten LOAD-Befehl gibt es natürlich noch die Möglichkeit, die Datei bereits beim Aufruf des Debuggers zu laden (z.B. DEBUG ASK.COM).

### 7.1.15 Programmentwicklung mit DEBUG

Alle bisher beschriebenen Befehle dienen mehr oder weniger zur Manipulation der Speicherinhalte. Mit DEBUG lassen sich aber auch Programme erstellen und testen. Hierfür stehen die Funktionen:

- ASSEMBLE
- LOAD
- NAME
- GO
- PROCEED
- TRACE
- UNASSEMBLE
- WRITE

zur Verfügung. NAME, LOAD und WRITE wurden bereits vorgestellt. Nachfolgend möchte ich auf die Technik zur Assemblierung kleinerer Programme eingehen.

### 7.1.16 Der ASSEMBLE-Befehl

Der Debugger besitzt einen zeilenorientierten Assembler, mit dem sich eingegebene Anweisungen direkt in Maschinencode umsetzen und im Speicher ablegen lassen. Die Funktion besitzt folgende Aufrufsyntax:

A [Seg:[Offs]]

Die Adresse gibt an wo der erste Befehl abgelegt werden soll. Fehlt die Adreßangaben, wird CS:0100 eingestellt. Alle Zahlen sind als Hexziffern einzugeben. Bei Kommandos zur Stringmanipulation ist der Datentyp explizit anzugeben (Byte oder Word). Ein Rücksprung (Return) über Segmentgrenzen ist als RETF (Return Far) einzugeben. Relative Sprünge werden automatisch mit dem korrekten Displacement assembliert, nachdem die absolute Zieladresse eingegeben wurde. Es ist jedoch auch möglich, einen entsprechenden Präfix (NEAR, FAR) einzugeben.

```
0100:0400 JMP 402           ; short jump
0100:0402 JMP NEAR 505     ; near jump
0100:0405 JMP FAR 50A      ; far jump
```

Der Präfix NEAR darf dabei mit NE abgekürzt werden. Bei Operationen auf Daten ist deren Typ (WORD, BYTE) explizit anzugeben.

```
MOVSB           ; Byte String Move
MOVSW           ; Word String Move
DEC WORD PTR [SI]
DEC BYTE PTR [SI]
```

Indirekte Adressierungen werden durch eckige Klammern markiert.

```
MOV AX,3F      ; lade AX direkt mit 3F
MOV AX,[OFF]   ; lade AX indirekt mit dem
                ; Wert der Adresse OFF
MOV AX,[SI]    ; lade indirekt mit dem
```

	; Wert der Adresse in SI
ADD BX,34[BP+2][SI+2]	; addiere indirekt
POP [BP+DI]	; Pop Memory Wert
PUSH [SI]	; Push Memory Wert

Die Pseudoinstruktionen DB (Byte) und DW (Wort) zur Eingabe von Konstanten werden unterstützt.

```
DB 1, 2, 3, "Hallo"
DW 3FF, 2000, 'Testtext'
DB "Hallo""Test"
```

Damit ist es möglich kleine Assemblerprogramme im Speicher zu erstellen. Mit ES:, DS: und CS: läßt sich die standardmäßige Segmenteinstellung für den nächsten Befehl überschreiben. Mit:

```
ES: MOV AX,[SI]
```

bezieht sich der Zeiger SI nicht auf das Daten- sondern auf das Extrasegment. Die Eingabe der Assembleranweisungen wird mittels der RETURN-Taste abgebrochen. Alle korrekt angegebenen Anweisungen werden Zeile für Zeile in den Speicher assembliert. Bei Fehleingaben zeigt DEBUG dies durch eine Fehlermeldung an:

```
-A CS:100
1400:0100 MOV AD,100
      ^Error
```

Dann wird die Adresse angezeigt und auf eine neue Eingabe gewartet. Auf diese Weise lassen sich beliebig lange Programme erstellen. Die einzige Einschränkung bei der Assemblierung betrifft die Verarbeitung von Labels. Labels werden von DEBUG nicht unterstützt, so daß Sprünge oder CALL-Befehle mit den direkten Adressen einzugeben sind. Falls das Sprungziel zur Eingabezeit noch nicht bekannt ist, kann eine Pseudokonstante (z.B. 0000) eingetragen werden.

```
JMP NEAR 0000
```

Der Assembler reserviert dann die korrekte Anzahl Opcodes im Speicher. Nachdem die Sprungadresse feststeht, lassen sich die entsprechenden Adressen entweder durch ENTER oder durch erneute Eingabe eines ASSEMBLE-Befehls an der jeweiligen Stelle nachtragen.

```
-A CS:100
1400:100 JMP NEAR 0122
```

### 7.1.17 Assemblierung aus einer Datei

Die direkte Eingabe von Assemblerbefehlen ist nur für kurze Programmtests handhabbar. Komfortabler ist es die Quelltexte aus einer Datei in einem Durchlauf zu übersetzen. Dies läßt sich in MS-DOS über einen Trick erreichen. Zuerst wird die Quelldatei mittels eines Texteditors erstellt. Diese darf beliebige (aber gültige) DEBUG-Anweisungen enthalten. Dann wird der Debugger unter Verwendung der DOS-I/O-Umleitung gestartet. Nachfolgend ist ein solcher Aufruf dargestellt:

```
DEBUG < ASK.ASM > ASK.LST
```

ASK.ASM steht für die Quelldatei, während in ASK.LST alle Meldungen während der Assemblierung gespeichert werden. Damit liest DEBUG die Eingaben nicht mehr von der Tastatur, sondern aus der Datei ASK.ASM. Die Textausgaben lassen sich weiter in die zweite Ausgabedatei (ASK.LST) umleiten. Nachfolgendes Beispiel erzeugt eine lauffähige COM-Datei zur Abfrage der DOS-Versionsnummer.

```
A CS:100
;-----
; Beispiel zur Assemblierung mit dem DOS-Debugger
; aus einer Textdatei.
;
; Aufruf:  DEBUG < VERSION.ASM > VERSION.LST
;
; In der Datei VERSION.LST werden alle Meldungen
; (auch Fehlermeldungen) des Debuggers abgelegt.
; Die lauffähige COM-Datei findet sich in
; VERSION.COM
;-----
; Assembliere den Programmcode ab CS:100
;
; ORG 100
;
MOV DX,0200          ; lade Adr. String 1
MOV AH,09           ; Ausgabe des Textes
INT 21              ; per INT 21
MOV AH,30           ; Abfrage der DOS-
INT 21              ; Version
MOV DL,30           ; Convert Main Nr.
ADD DL,AL           ;
MOV AH,02           ; Ausgabe Character
INT 21              ;
MOV DL,2E           ; write "."
INT 21              ;
MOV DL,30           ; Convert Second Nr.
ADD DL,AH           ;
INT 21              ; write char.
MOV DX,0210         ; lade Adr. String 2
MOV AH,09           ; Ausgabe CR,LF
INT 21              ; per INT 21
MOV AX,4C00         ; Terminate Process
INT 21              ; normal
NOP                 ; Ende des Programmcodes
```

```
; hier muß eine Leerzeile folgen !!!  
  
A CS:200  
;  
; ORG 200  
;  
; Assembliere die Text-Konstanten ab CS:200  
; statischer Text  
DB "MS-DOS Version $"  
; CR,LF  
DB 0A,0D,"$"  
;-----  
; END      Leerzeile beendet Assemble Mode  
;-----  
; speichere den Code in der Datei VERSION.COM  
;-----  
; hier muß eine Leerzeile folgen  !!  
  
N VERSION.COM  
R BX  
0  
R CX  
200  
W CS:100  
Q
```

*Listing 7.1: VERSION.ASM*

Interessant ist in diesem Zusammenhang noch eine undokumentierte Eigenschaft von DEBUG. Normalerweise lassen sich in DEBUG keine Kommentare eingeben. Ist jedoch der ASSEMBLE-Befehl aktiv, akzeptiert DEBUG Kommentarzeilen. Alle Texte mit einem vorangestellten Semikolon werden als Kommentar betrachtet und überlesen. Lediglich bei DB- und DW-Anweisungen erfolgt eine Fehlermeldung, so daß DEBUG hier keine Kommentare akzeptiert. In obigem Programm bewirkt die erste "A CS:100" Anweisung, daß DEBUG den Kommentarkopf überliest. Die Daten werden in den Bereich ab CS:200 assembliert. Vor der Anweisung "A CS:200" muß eine Leerzeile stehen, um mit dem RETURN-Zeichen der Leerzeile das noch aktive A-Kommando abubrechen. Wichtig ist auch, daß in einem Block mit Assembleranweisungen keine Leerzeilen auftreten, da diese auch den ASSEMBLE-Mode beenden.

Mit der W-Anweisung lassen sich 200H Byte ab CS:100 speichern. Die Sequenz:

```
N VERSION.COM  
R BX  
0  
R CX  
200  
W  
Q
```

ist deshalb in jedem Quellprogramm, getrennt durch eine Leerzeile, an den Assemblercode anzuhängen. Ohne den Q-Befehl hängt sich DEBUG auf, da das Programm ja alle Anweisungen aus der Quelldatei erwartet. Hier muß folglich als letztes auch die Abbruchanweisung stehen. Die restlichen Befehle legen eine COM-Datei an. Ist die Länge des Programmes noch nicht bekannt, kann in den Registern CX und BX ein Pseudowert abgelegt werden. Nach der Übersetzung liegt die Programmlänge vor und kann in den Quelltext eingesetzt werden. Dann ist das Programm erneut zu übersetzen.

Nach der Assemblierung stehen alle DEBUG-Meldungen (auch Fehlertexte) in der Datei VERSION.LST. Der Inhalt dieser Datei wird nachfolgend angezeigt.

```

-A CS:100
29E9:0100 ;-----
29E9:0100 ; Beispiel zur Assemblierung mit dem DOS-Debugger
29E9:0100 ; aus einer Textdatei.
29E9:0100 ;
29E9:0100 ; Aufruf:  DEBUG < VERSION.ASM > VERSION.LST
29E9:0100 ;
29E9:0100 ; In der Datei VERSION.LST werden alle Meldungen
29E9:0100 ; (auch Fehlermeldungen) des Debuggers abgelegt.
29E9:0100 ; Die lauffähige COM-Datei findet sich in
29E9:0100 ; VERSION.COM
29E9:0100 ;-----
29E9:0100
-A CS:200
29E9:0200 ;
29E9:0200 ; ORG 200
29E9:0200 ;
29E9:0200 ; Assembliere die Text-Konstanten ab CS:200
29E9:0200 ; statischer Text
29E9:0200 DB "MS-DOS Version $"
29E9:0210 ; CR,LF
29E9:0210 DB 0A,0D,"$"
29E9:0213 ;
29E9:0213 ; Assembliere den Programmcode ab CS:100
29E9:0213 ;
29E9:0213
-A CS:100
29E9:0100 ;
29E9:0100 ; ORG 100
29E9:0100 ;
29E9:0100 MOV DX,0200           ; lade Adr. String 1
29E9:0103 MOV AH,09           ; Ausgabe des Textes
29E9:0105 INT 21              ; per INT 21
29E9:0107 MOV AH,30           ; Abfrage der DOS-
29E9:0109 INT 21              ; Version
29E9:010B MOV DL,30           ; Convert Main Nr.
29E9:010D ADD DL,AL           ;
29E9:010F MOV AH,02           ; Ausgabe Character
29E9:0111 INT 21              ;
29E9:0113 MOV DL,2E           ; write "."
29E9:0115 INT 21              ;
29E9:0117 MOV DL,30           ; Convert Second Nr.
29E9:0119 ADD DL,AH           ;
29E9:011B INT 21              ; write char.

```

```

29E9:011D MOV DX,0210          ; lade Adr. String 2
29E9:0120 MOV AH,09           ; Ausgabe CR,LF
29E9:0122 INT 21              ; per INT 21
29E9:0124 MOV AX,4C00         ; Terminate Process
29E9:0127 INT 21              ; normal
29E9:0129 NOP                 ; Ende des Programmcodes
29E9:012A ;-----
29E9:012A ; END      Leerzeile beendet Assemble Mode
29E9:012A ;-----
29E9:012A ; speichere den Code in der Datei VERSION.COM
29E9:012A ;-----
29E9:012A
-N VERSION.COM
-R BX
BX 0000
:0
-R CX
CX 0000
:200
-W
Schreibe 0200 Bytes
-Q

```

Listing 7.2: VERSION.LST

Die Umleitung der DEBUG-Ausgaben läßt sich mit:

```
DEBUG < VERSION.ASM
```

auf den Bildschirm leiten. Wird als Ausgabeinheit PRN: angegeben, erzeugt DEBUG sofort ein Listing auf dem Printer.

### 7.1.18 Der UNASSEMBLE-Befehl

Als weitere Funktion bietet DEBUG einen eingebauten Disassembler. Diese Funktion besitzt folgende Aufrufsyntax:

```
U [adresse] [range]
```

Als Segmentadresse wird normalerweise CS angenommen. Die Startadresse liegt bei CS:100, wenn nichts anderes spezifiziert wurde. Sofern Sie den Code aus dem ersten Beispiel (Versionsabfrage im Assemblemodus) ab Adresse CS:100 assembliert haben, geben sie den Befehl ein:

```
-U 100 127
```

Auf dem Bildschirm wird in etwa folgende Ausgabe erscheinen:

```

1007:0100 BA0001      MOV  DX,0100
1007:0103 B409       MOV  AH,09

```

```
1007:0105 CD21      INT    21
1007:0107 B430      MOV    AH,30
.....
```

Obiger Ausdruck enthält die rückübersetzten Maschinencodebefehle des Programmes im angegebenen Adressbereich.

Wird eine Länge eingegeben, versucht UNASSEMBLE mindestens die Zahl der Bytes zu bearbeiten. Ohne Längenangaben werden ca. 20 Bytes bearbeitet. Die genaue Länge richtet sich aber nach dem Befehlstyp, da nur vollständige Befehle decodiert werden. Die Startadresse muß auf einer gültigen Befehlsadresse liegen, da sonst ungültige Ausgaben erzeugt werden. Fehlt die Startadresse, beginnt die Disassemblierung mit der durch das Register IP spezifizierten Adresse. Es soll aber noch eine weitere Besonderheit erwähnt werden. Geben Sie bitte den folgenden Text ein:

```
-E CS:100 80 00 55 82 00 55
-A 106
1007:0106 JC 100
1007:0108 JNE 100
1007:010A JNA 100
1007:010C
```

Der UNASSEMBLE-Befehl:

```
-U 100 10C
1007:0100 800055      ADD    BYTE PTR [BX + SI],55
1007:0103 820055      ADD    BYTE PTR [BX + SI],55
1007:0106 72F8        JB     0100
1007:0108 75F6        JNZ   0100
1007:010A 76F4        JBE   0100
-
```

bringt ein merkwürdiges Ergebnis. Obwohl zwei verschiedene Opcodes 80 und 82 eingegeben wurden (siehe Codes vor den Befehlen), zeigt der Disassembler in beiden Fällen einen identischen (ADD) Befehl an. Die assemblierten Sprungbefehle werden auch anders wiedergegeben. Die Entwickler des 8086 haben den gleichen Befehl unter mehreren Opcodes implementiert. Daher taucht der ADD-Befehl zweifach auf. Die falschen Sprungbefehle finden ebenfalls eine Erklärung. INTEL erlaubt für eine Abfrage mehrere Mnemonics (symbolische Assemblerbefehle). So sind z.B. JB, JNAE, JC drei gültige Sprungbefehle mit der gleichen Bedingung, die nur unterschiedlich formuliert wurde. Der Assembler akzeptiert alle drei Anweisungen, setzt sie aber in einen Opcode um. Der Disassembler kann nun nicht mehr erkennen, welche Bedingung ursprünglich formuliert wurde. Er wird daher immer nur eine Bedingung "JC" zurückgeben. Bei den SHIFT-Befehlen sind in DEBUG nicht alle Varianten implementiert. Der Disassembler ist trotzdem sehr hilfreich um Programmbereiche in die entsprechenden Assemblerbefehle zurückzuwandeln.

### 7.1.19 Programmtests mit DEBUG

Nachdem nun alle Techniken zur Programmentwicklung, zum Laden und Speichern von Daten, sowie die Befehle zur Speicherbearbeitung besprochen sind, kommen wir zum letzten Punkt. Was noch fehlt ist die Kenntnis, wie der Debugger zum Programmtest einzusetzen ist. Hierzu bietet DEBUG durchaus einige Funktionen.

### 7.1.20 Der GO-Befehl

Normalerweise ist es notwendig, die Programme zum Test unter der Kontrolle von DEBUG ablaufen zu lassen. Nur dann ist ein Test möglich. Für diesen Zweck stellt DEBUG die GO-Funktion zur Verfügung. Die allgemeine Aufrufsyntax lautet:

```
G [=adress] [adress] [adress] ..
```

Wird keine Adresse eingegeben, übernimmt GO den Inhalt der CS und IP-Register und startet den Programmablauf. Damit verliert DEBUG die Kontrolle über die Programmausführung. Durch Eingabe der optionalen Startadresse (markiert durch ein vorangestelltes Gleichheitszeichen):

```
G = 100
```

benutzt der GO-Befehl diese als Startadresse. Dabei gelten die üblichen Eingabekonventionen. Fehlt die Segmentangabe, übernimmt DEBUG den Inhalt des CS-Registers. Wichtig ist die Eingabe des Gleichheitszeichens vor der ersten Adresse. Damit läßt sich der Programmablauf in jedem beliebigen Schritt starten. Es ist darauf zu achten, daß an der spezifizierten Adresse ein gültiger Befehl beginnt, da sonst undefinierte Effekte auftreten. Mit:

```
-G =100 127
```

wird ein Teil des Programmes ausgeführt und ab der Adresse 127 zu DEBUG zurückgekehrt. Offenbar setzt DEBUG im Programm Unterbrechungspunkte. Wird kein Haltepunkt erreicht und DEBUG meldet:

```
Programm terminated normally
```

dann muß das Programm vor der nächsten Ausführung erneut geladen werden. Ohne Segmentangabe übernimmt der GO-Befehl den Inhalt des CS-Register. Mit der Eingabe:

```
-G 113 126
```

werden zwei Haltepunkte im Codesegment gesetzt. Die Programmausführung beginnt mit den aktuell gesetzten Werten im CS:IP Register, da die Option *=adress* nicht verwendet wurde.

### 7.1.21 Der TRACE-Befehl

Als weitere Funktion stellt DEBUG noch den TRACE-Befehl zur Verfügung. Das Aufrufformat lautet:

T [= adress] [value]

TRACE beginnt mit der Abarbeitung des Programmcodes ab der angegebenen Adresse in Form von Einzelschritten, wobei nach jedem Schritt die Registerinhalte und der nächste Befehl angezeigt werden. Wird eine Adresse angegeben, beginnt der Programmablauf bei dieser Adresse. Bei fehlender Segmentangabe übernimmt DEBUG den Inhalt des CS-Registers. *Value* gibt an, wieviele Einzelschritte durchzuführen sind. Mit:

-T = 100

wird z.B. das erste Beispielprogramm zur Versionsabfrage gestartet und der Inhalt der Register wird nach Ausführung des ersten Befehls gezeigt. Mit:

-T 10

werden weitere 16 Schritte (10H) ausgeführt. Da TRACE alle Hardware Interrupts vor Ausführung eines Befehls sperrt, darf das Benutzerprogramm die Interruptmaske des 8295 Controllers nicht verändern. Andernfalls sind unkontrollierbare Effekte möglich. Wird im Anwenderprogramm der INT 3 (Trap) benutzt, setzt TRACE auf den INT 3 Code einen Unterbrechungspunkt.

### 7.1.22 Der PROCEED-Befehl

Enthält ein Programm viele Unterprogrammaufrufe, ist es störend, daß mit TRACE jedesmal auch der Code des Unterprogrammes mit angezeigt wird. Beim Aufruf der DOS-INT 21-Funktionen mit TRACE kann es sogar zu Systemabstürzen kommen. Hier bietet die PROCEED-Funktion Hilfestellung. Der Aufruf:

P [Adresse] [Wert]

führt dazu, daß bei Unterprogrammaufrufen, Interrupts oder String-Move-Befehlen der Programmablauf erst nach der Ausführung unterbrochen wird. Bei anderen Befehlen verhält sich PROCEED wie die TRACE-Option. Nach Ausführung der Instruktion oder des Unterprogrammes läßt sich dann der Registerinhalt untersuchen. Außerdem wird eine Wiederholung des Befehls möglich. Die Adreßkonventionen entsprechen

den bereits behandelten Regeln. Wert gibt an, wie oft eine Funktion wiederholt wird, bevor eine Unterbrechung auftritt. Allerdings ist dieser Befehl nicht bei allen DEBUG-Versionen implementiert.

### 7.1.23 Die Expanded-Memory-Befehle

DEBUG besitzt ab DOS 4.0 einige Befehle um auf das Expanded-Memory zuzugreifen. Diese Befehle möchte ich aber hier nicht vorstellen. Gegebenenfalls ist die Originalliteratur oder /2/ zu konsultieren.

### 7.1.24 Anmerkungen zu DR-DOS 5.0/6.0

DR-DOS besitzt das Programm DEBUG.COM nicht, vielmehr wird der Debugger SID.EXE mitgeliefert. Dieses Programm weicht in der Kommandosyntax bei den Assembleranweisungen etwas von DEBUG ab. Die wesentlichen Änderungen beim Debugger betreffen die Kommandosyntax: SID ist eine bildschirmorientierte Version. Der Aufruf erfolgt mit:

```
SID <Parameter>
```

Als Parameter lassen sich Dateinamen und weitere Optionen angeben. Die DOS-Ein-/Ausgabeumleitung kann wie bei DEBUG verwendet werden. Nach dem Start meldet sich SID mit:

```
-----
*** Symbolic Instruction Debugger ***  Release 3.1
      Copyright (c) 1983,1984,1985,1988,1990
      Digital Research, Inc. All Rights Reserved
-----
```

```
#
```

Als PROMPT wird das Zeichen # benutzt; beendet wird SID mit dem Kommando Q. Die Befehle des Debuggers lassen sich mit:

```
#?
```

abrufen. Auf dem Bildschirm erscheint eine kurze Meldung mit einer Auflistung der möglichen Eingabebefehle. Neben dem Fragezeichen zur Aktivierung der Online-Hilfe stehen verschiedene Befehle zur Verfügung. Für eine genaue Beschreibung sind die DR-DOS Unterlagen zu konsultieren.

Mit der Eingabe ?? läßt sich die genaue Syntax der einzelnen Befehle am Bildschirm abfragen. Leider hat SID aus Sicht des Assemblerprogrammiers einen weiteren Nachteil, die Kommandos sind nicht ganz kompatibel zu DEBUG.COM. Deshalb müssen die Listings für DR-DOS in einigen Punkten angepaßt werden. Nachfolgend möchte ich die wichtigsten Änderungen stichpunktartig vorstellen:

- ◆ Alle Kommentare sind aus dem Listing zu entfernen.
- ◆ Alle impliziten JMP-Anweisungen (z.B. JMP 100) müssen in SID mit einem Prefix (Short) versehen werden (z.B. JMP SHORT 100 wird zu JMPS 100).
- ◆ Die Assemblierung muß mit Axxx beginnen (z.B. A100).
- ◆ Die Länge des Programmcodes ist in SID mit der Anweisung:  
*Filename.COM Start Länge* abzuspeichern.

Mit *Filename* ist der Dateiname der zu erzeugenden COM-Datei gemeint. Der Parameter *Start* enthält die Offsetadresse ab der der Programmcode beginnt. In *Länge* wird die Zahl der zu speichernden Bytes definiert. Nach diesen Modifikationen sollten sich obige Programme übersetzen lassen. Falls bei weiteren Befehlen doch Probleme auftreten konsultieren Sie bitte die DR-DOS Handbücher.

**Anmerkung:** In Novell DOS 7.0 besitzt der Debugger die gleiche Syntax wie das DOS-Pendant.

# Anhang A

## Batchprogramm zur Verwendung von ASK.COM

```

ECHO OFF
:=====
: Programm: ASM.BAT
: Version: 1.0 (c) Born
: Batchdatei zur Demonstration von ASK, Steuerung des Über-
: setzervorganges bei A86. Bei Aufruf ohne Parameter wird ein
: Hilfstext angezeigt.
:=====
: Filename als Parameter eingegeben ?
IF '%1' == '' GOTO EXIT1
: LOOP      -> Benutzermeldung ausgeben
CLS
ECHO -----
ECHO !      Utility zur Steuerung der Assemblerentwicklung      !
ECHO !-----!
ECHO !
ECHO !                      0 Exit                                !
ECHO !                      1 Edit                                !
ECHO !                      2 Assemble                             !
ECHO !                      3 Display File                         !
ECHO !                      4 Print File                          !
ECHO !
ECHO !-----!
ECHO.
ASK Bitte geben Sie einen Code ein ?:
: Teste den von ERRORLEVEL zurückgegebenen Code
: Ziffern > 4 (ASCII Code ab 53) abfangen
IF ERRORLEVEL 53 GOTO LOOP
IF ERRORLEVEL 52 GOTO L4
IF ERRORLEVEL 51 GOTO L3
IF ERRORLEVEL 50 GOTO L2
IF ERRORLEVEL 49 GOTO L1
IF ERRORLEVEL 48 GOTO EXIT
: Code kleiner 0 abfangen
GOTO LOOP
:L1      -> Aufruf des Editors
EDIX %1.ASM
GOTO LOOP
:L2      -> Aufruf des A86 für COM-Datei
A86 %1.ASM
GOTO LOOP
:L3      -> Ausgabe des ASM-Files
TYPE %1.ASM | MORE
GOTO LOOP
:L4      -> Ausgabe auf Printer
PRINT %1.ASM
GOTO LOOP
: EXIT1   -> Fehlermeldung
ECHO !!! Bitte Filename ohne Extension !!!

```

```
ECHO !!! z.B.      ASM NUMOFF
ECHO !!!      beim Aufruf mit angeben      !!!
:EXIT
ECHO ON
```

# Anhang B

Batchprogramm zur Einstellung der Schriftarten am Drucker mit ESC.EXE.

```

ECHO OFF
:=====
: Programm: XPRINT.BAT
: Version: 1.2 (c) Born
: Batchdatei zur Formatumstellung eines Druckers. Bei Auf-
: ruf ohne Parameter wird ein Hilfstext angezeigt. Sonst
: stellt das Programm den Drucker um.
:=====
:LOOP
CLS
ECHO -----
ECHO ! Utility zur Umstellung der Schriftart des Druckers !
ECHO -----
ECHO !                               Formate                               !
ECHO !                               !                               !
ECHO !                               6 Draft                               !
ECHO !                               5 Pica                               !
ECHO !                               4 Roman                               !
ECHO !                               3 Sans Serif                          !
ECHO !                               2 Kursiv Ein                          !
ECHO !                               1 Kursiv Aus                          !
ECHO !                               0 Exit                               !
ECHO !                               !                               !
ECHO -----
ECHO.
: prüfe Eingabeparameter
ASK Bitte Code eingeben :
IF ERRORLEVEL 55 GOTO EXIT
IF ERRORLEVEL 54 GOTO L0
IF ERRORLEVEL 53 GOTO L1
IF ERRORLEVEL 52 GOTO L2
IF ERRORLEVEL 51 GOTO L3
IF ERRORLEVEL 50 GOTO L4
IF ERRORLEVEL 49 GOTO L5
GOTO EXIT
:L0
ECHO Draft Modus einschalten
ESC 1B 78 00 > PRN:
GOTO EXIT
:L1
ECHO Pica Modus einschalten
ESC 1B 21 00 > PRN:
GOTO EXIT
:L2
ECHO Roman Modus einschalten
ESC 1B 78 01 1B 6B 00 > prn:
GOTO EXIT
:L3
ECHO Sans Serif Modus einschalten

```

```
ESC 1B 78 01 1B 6B 01    > prn:
GOTO EXIT
:L4
ECHO Kursiv Schrift einschalten
ESC 1B 34    > prn:
GOTO EXIT
:L5
ECHO Kursiv Schrift ausschalten
ESC 1B 35    > prn:
:EXIT
ECHO ON
```

# Anhang C

## Debuggen mit dem D86

Mit dem A86-Assembler wird ein symbolischer Debugger D86 mitgeliefert. Mit ihm lassen sich die Programme schrittweise austesten. Die Dokumentation wird auf der Diskette mitgeliefert. Nachfolgend möchte ich kurz auf die Möglichkeiten des Werkzeuges eingehen.

Der Aufruf des D86 erfolgt mit der Syntax:

```
D86 [+V] Filename [Parameter]
```

Mit dem Schalter läßt sich der virtuelle Debug-Modus mit zwei Bildschirmen (Mono und Color) aktivieren. Näheres findet sich in der D86 Dokumentation.

Hinter dem Dateinamen lassen sich Parameter für das aufgerufene Programm angeben. Diese werden dann im PSP ab Offset 80H im Puffer abgelegt. Der D86 sucht den File mit den Extensionen .COM oder .EXE, falls diese nicht explizit angegeben wurden. Zudem wird eine Datei mit dem Namen und der Extension .SYM gesucht und gegebenenfalls geladen. Diese Datei enthält die Symbole (Variablennamen, Labels, etc.), die der D86 im Disassemblermodus anzeigt. Fehlt der SYM-File, werden die Informationen nicht eingeblendet. Probleme gibt es allerdings, falls die Module separat assembliert werden, dann werden die Symbolfiles dieser Module in der Regel nicht geladen.

Nach dem Start meldet sich D86 mit einem Fenster, das die ersten Anweisungen des Programmes und den Status der Register zeigt. In der rechten Hälfte sind noch einige Copyright Informationen zu sehen. Mit den Tasten ALT-F10 lassen sich Hilfinformationen abrufen und mit F10 gelangt man jederzeit in das Fenster zurück. Nach dem Aufruf erlaubt der D86 Kommandos:

- ◆ zur direkten Assemblierung von Kommandos
- ◆ zum Starten und Test des Programmes als Befehlseingaben
- ◆ Anweisungen über CTRL- und Funktionstasten
- ◆ Die Zahlen 1-6, um die Display Windows zu adressieren
- ◆ Ausführung einzelner Anweisungen

Die einzelnen Modi sind in der Dokumentation beschrieben und sollen kurz **ang**eprochen werden.

## Der A86-Assembler-Modus [F7]

Mit der Funktionstaste F7 läßt sich der Patch-Modus **akt**ivieren, d.h., die A86-Kommandos lassen sich direkt eingeben und werden sofort in ausführbare Anweisungen übersetzt. Mit diesen Anweisungen lassen sich auch die Segmentregister laden oder Programmsprünge ausführen. Die Anweisungen werden direkt an den aktuellen Adressen in den Speicher geschrieben (gepatcht). Mit DW oder DB lassen sich auch einzelne Werte direkt als Hexzahlen in den Speicher schreiben. Mit der EQU-Directive lassen sich Symbole definieren, die beim Debuggen die Symboltabelle ergänzen.

Mit den folgenden Funktionstasten lassen sich einzelne Funktionen des D86 aktivieren:

- ALT-F10: Mit diesen Tasten wird der Hilfemodus aufgerufen, der die Tastenbelegung angibt.
- F10: Mit dieser Taste läßt sich zwischen verschiedenen Bildschirmfenstern hin- und herschalten.
- CTRL-I: Schaltet von einem Fenster direkt in die Darstellung des Disassemblers.
- CTRL-S: Schaltet sofort in das D86-Status-Fenster.
- CTRL-F: Schaltet den Floating-Point Mode ein (nur bei (8087/80287-Chips im PC).
- CTRL-N: Schaltet aus dem Memory-Display zum nächsten Fenster.
- CTRL-P: Mit Previous Page läßt sich der Zeiger auf die vorherige Seite zurücksetzen.
- CTRL-Q: Mit diesen Tasten läßt sich das letzte Hilfsfenster wieder aktivieren.

Die folgenden Tasten positionieren den Instruction Pointer. Mit Cursor Down, CTRL-D, Cursor Up, CTRL-U läßt sich der IP um eine Anweisung nach oben oder unten **sch**reiben. Mit PgUp und PgDn wird der IP seitenweise innerhalb des Programmes bewegt. Mit Home springt der D86 an den Programmanfang und mit CTRL-E an das Programm**ende**. Dies ist hilfreich, falls der Programmablauf an bestimmten Stellen gestartet werden soll.

Mit den Funktionstasten F1, F2, F4 und F6 läßt sich das Programm ausführen. Dabei gilt folgende Belegung:

- F1: Führt das Programm ab der aktuellen Adresse im Einzelschritt aus, d.h., die folgende Anweisung wird abgearbeitet und dann die Registerbelegung angezeigt. Mit einem weiteren Druck auf F1 ist die nächste Anweisung ausführbar. Die Funktion sollte nicht bei externen INT- und CALL-Aufrufen verwendet werden.
- F2: Diese Taste aktiviert den Prozedur-Mode. Einzelanweisungen werden wie im F1-Mode behandelt. Bei INT- und CALL-Aufrufen wird jedoch die komplette Prozedur ohne Unterbrechung ausgeführt.
- F4: Erlaubt die Ausführung einer bedingten Sprunganweisung und unterbricht an der Sprungmarke. Bei normalen Befehlen wird die F1-Option benutzt.
- F6: Startet das Programm und setzt eine Unterbrechungsadresse auf dem Stack ab.

Außerdem besitzen die folgenden Funktionstasten eine erweiterte Bedeutung:

- F3: Wiederholt die letzte eingegebene Assembler- oder Debuggeranweisung.
- F7: Aktiviert den Patch Mode, in dem sich Assembleranweisungen eingeben lassen. Mit SHIFT-F7 wird die aktuelle CS:IP Adresse für Eingaben im F-Mode markiert.

Verschiedene Befehle lassen sich im Debug-Kommando-Modus eingeben:

- B: Setzt oder löscht eingetragene Unterbrechungspunkte im Programm. Die Unterbrechungspunkte lassen sich mit dem G-Kommando setzen. Mit B <CR> werden die Unterbrechungspunkte gelöscht. Werden hinter B Adressen angegeben, setzt das Kommando die Unterbrechungen (z.B. B,Exit).
- D: Setzt oder löscht einen Unterbrechungspunkt im Datenbereich. Das Kommando ist nur auf 80386-Maschinen verfügbar.
- F: Mit diesem Kommando läßt sich ein String innerhalb des Speichers suchen. Die Suche beginnt ab CS:IP, die mit SHIFT-F7 markiert werden kann.
- G: Mit diesem Kommando wird die Ausführung des Programmes ab der aktuellen Adresse gestartet. Hinter dem Kommando lassen sich Adressen für Unterbrechungen eingeben.
- J: Mit diesem Kommando verzweigt D6 zu dem angegebenen Operanden im aktuellen Codesegment.
- L: Mit diesem Kommando wird ein disassembliertes Listing auf dem Drucker ausgegeben. Mit L,0200,LIST.LST läßt sich ein Bereich, beginnend ab CS:IP, mit einer Länge von 512 Byte in die Datei LIST.LST disassemblieren.
- O: Setzt den INT 21 Breakpoint, d.h., jedesmal wenn ein INT 21 aktiviert wird, gibt D86 den Inhalt des AH-Registers aus. Das Kommando erlaubt zwei Parameter, welche die Grenzen der Codes für AH angeben. Mit O,030 wird nur die Funktion 30H beim Aufruf angezeigt.
- Q: Mit diesem Kommando läßt sich der D86 beenden.
- W: Mit diesem Kommando kann ein Teil des Programmes aus dem Speicher in eine Datei geschrieben werden.

Weitere Kommandos lassen sich zum Anzeigen von Daten aus dem Speicher anzeigen.

- @: Mit diesem Kommando wird das folgende Byte als Zähler genommen und ein String der entsprechenden Länge aus den folgenden Bytes ausgegeben.
- B: Zeigt den Inhalt eines einzelnen Bytes als Hexwert an.
- C: Zeigt den Inhalt eines Bytes als ASCII-Zeichen an.
- D: Zeigt den Inhalt eines 16-Bit-Wortes als Hexzahl an.
- F: Zeigt den Inhalt des Speichers als Kommazahl an.
- N: Zeigt ein Byte als vorzeichenlose Zahl an.
- O: Zeigt ein Word als Oktalwert an.
- Q: Zeigt ein Byte als Oktalwert an.
- L: Zeigt einen Textstring im C-Format an.
- S: Zeigt einen ASCII-Z-String (Abschluß mit 00H) an.
- W: Zeigt den Inhalt einer 16-Bit-Zahl als Word an.

Um den Umfang nicht allzu sehr auszudehnen, wurden die D86-Befehle nur kurz angesprochen. Weitere Informationen finden sich in der Dokumentation des Produktes.

# Anhang D

## Die Fehlermeldungen des A86

### *~01 Unknown Mnemonic~*

Der A86 hat eine Anweisung gefunden, die nicht zu den gültigen Befehlen gehört.

### *~02 JUMP > 128~*

Es wurde versucht, einen Bereich von mehr als 128 Byte mit einem SHORT-Sprung zu überspringen.

### *~03 [BX+BP] And [SI+DI] Not Allowed~*

Die benutzte 8086-Anweisung erlaubt nicht die Verwendung dieser Indexregister.

### *~04 Bad Character In Number~*

Alle Zahlen müssen mit einer Ziffer beginnen und dürfen auch nur gültige Ziffern enthalten. Die Zahl enthält eine ungültige Ziffer.

### *~05 Operands Not Allowed~*

Bei diesem Befehl sind keine Operanden erlaubt (z.B.: PUSHF, STOSB, STC, FLDPI, CLTS).

### *~06 Symbol Required~*

In der Anweisung wird ein Symbol erwartet. Die Fehlermeldung kann eine Reihe von Ursachen haben (s. Original-A86-Fehlerliste).

### *~07 Local Symbol Required~*

Dieser Fehler wird angezeigt, falls in einem JMP-Befehl ein allgemeines globales Label das Zeichen > angegeben wird. Mit JMP >Lx darf nur auf ein lokales Label mit x = 0 bis 9 verzweigt werden.

### *~08 Too Many Operands~*

Diese Meldung tritt bei Befehlen und Direktiven auf, falls die Zahl der angegebenen Parameter größer als die Zahl der erwarteten Parameter ist. Manchmal ist ein überzähliges Komma die Ursache für diese Meldung.

*~09 Constant Required~*

In der betreffenden Anweisung wird eine Konstante erwartet. Der Fehler wird bei Befehlen und Direktiven (ENTER, RET, RADIX, etc.) angegeben, falls keine Konstante folgt. Das gleiche gilt für Ausdrücke (\*, /, SHL, OR, NOT, BY, etc.).

*~10 More Operands Required~*

Die Meldung wird erzeugt, falls ein Operator zwei Operanden benötigt, aber keiner oder nur ein Operand angegeben wurde. Ursache ist häufig ein falsch gesetztes Komma.

*~11 Constant/Label Not Allowed~*

In der betreffenden Anweisung ist eine Konstante oder ein Label nicht erlaubt (z.B. Zuweisung MOV 03FFF,AX).

*~12 Segment Register Not Allowed~*

Es wurde ein Segment-Register in einem Befehl verwendet, obwohl für diesem Befehl keine Segmentregister erlaubt sind.

*~13 Byte/Word Combination Not Allowed~*

Die Kombination von Bytes oder Worten ist nicht erlaubt. Der Fehler tritt bei 2-Byte-Anweisungen auf, falls ein Operand als Byte und der andere als Word definiert ist.

*~14 Bad Operand Combination~*

Die Kombination der Operanden ist nicht erlaubt (z.B. DT 3.7+BX). Ferner tritt der Fehler auch auf, falls zwei Operanden unterschiedliche Größen besitzen.

*~15 Bad Subtraction Operands~*

Der Fehler tritt bei dem Versuch auf, Ausdrücke von Operanden zu subtrahieren, die keine Subtraktion erlauben.

*~16 Definition Conflicts With Forward Reference~*

Die verwendete Vorwärtsreferenz kann durch den A86 in diesem Fall nicht aufgelöst werden.

*~17 Divide Overflow~*

Bei der Division ist ein Überlauf aufgetreten. Dies kommt vor, wenn eine Berechnung innerhalb eines Ausdrucks den 64 KByte Bereich verläßt.

*~18 Same Type Required~*

Bei Vergleichsoperatoren (EQ, NE, GT, GE, LT, or LE) benötigen die Operanden den gleichen Datentyp.

*~19 CS Destination Not Allowed~*

Der Fehler tritt beim Versuch auf, das CS-Register als Zielregister bei MOV oder POP zu nutzen. Gültig sind nur JMP, CALL und RET-Anweisungen, um CS zu verändern.

*~20 Left Operand Not Allowed~*

Der Operand der linken Seite des Ausdrucks ist ungültig. Solche Operatoren sind BIT, NOT, OFFSET, TYPE, LOW, HIGH, SHORT, LONG, und INT.

*~21 Bad Single Operand~*

Der angegebene Operand ist für die zugehörige Instruktion nicht zulässig.

*~22 Bad DUP Usage~*

Tritt auf, falls eine DUP Anweisung außerhalb des Kontext gerät.

*~23 Number Too Large~*

Die angegebene numerische Konstante ist zu lang, um durch den A86 verarbeitet zu werden. (Limit Integer =  $2^{**}80-1$ ).

*~24 SEGMENT or ENDS Required~*

Tritt auf, wenn nach den Schlüsselworten CODE oder DATE kein Schlüsselwort SEGMENT oder ENDS gefunden wird.

*~25 Bad CALL/JMP Operand~*

Der Operand für die CALL- oder JMP-Anweisung ist unzulässig. Tritt auch bei der Übersetzung in OBJ-Files auf, falls das Sprungziel als Konstante angegeben wird.

*~26 Memory Doubleword Required~*

Es wird ein 4-Byte-DWORD erwartet (LDS, LES, BOUND).

*~27 Bad IN/OUT Operand~*

Der Operand für den IN/OUT-Befehl besitzt nicht die korrekte Form.

*~28 :type Required~*

In der EXTRN-Liste wird explizit die Definition eines Typs für den Aufruf gefordert.

*~29 Bad Rotate/Shift Operand~*

Der Fehler tritt auf, falls der Zähler hinter dem 2. Byte größer als 1 wird und keine Variable ist.

*~30 Byte-Sized Constant Required~*

Es wird eine absolute Bytekonstante erwartet (z.B. Operanden in BIT-Vergleichen).

*~31 Instruction In Data Segment Not Allowed~*

Die Anweisung ist im DATA-Segment nicht zulässig. In STRUC sind die Directiven DB, DW, DD, DQ, DT, STRUC, ENDS, EQU, SEGMENT, GROUP, MACRO, LABEL, EVEN, und ORG erlaubt. Im DATA Segment sind die gleichen Directiven sowie PROC, ENDP, DATA, und CODE erlaubt.

*~32 Bad String~*

Der String ist nicht mit einem Hochkomma abgeschlossen.

*~33 Bad Data Operand~*

Es wurde ein ungültiger Operator in der Data-Initialisierungs-Anweisung (DB, DW, etc.) spezifiziert.

*~34 Index Brackets Required~*

Tritt auf, falls ein Register bei indirekten Zugriffen nicht in eckigen Klammern gesetzt wurde.

*~35 Bad Character~*

Das Zeichen ist für den A86 nicht verarbeitbar. Dies sind am Zeilenbeginn: Ziffern, / - , + \* () & " !.

*~36 String > 2 Not Allowed~*

Der String besitzt mehr Zeichen als die Anweisung aufnehmen kann (z.B. 3 Zeichen in DB-Anweisung).

*~37 Misplaced Built-In Symbol~*

Das Symbol vor der Fehlermeldung ist durch den A86 belegt.

*~38 Segment Combination Not Allowed~*

Die Segmentkombination ist nicht erlaubt.

*~39 Bad Index Register~*

Das angegebene Register darf nicht als Index verwendet werden. Erlaubt sind: SI, DI, BX, BP.

*~40 Conflicting Multiple Definition Not Allowed~*

Eine Mehrfachdefinition ist nicht erlaubt, kommt aber vor. So können Labels mit dem gleichen Namen mehrfach auftreten. Bei EQU sollte die = Anweisung zur Redefinition verwendet werden.

*~41 ENDS Has No Segment~*

Die ENDS-Anweisung kann keinem Segment zugeordnet werden.

*~42 Bad IF Operand~*

Der IF-Operand ist ungültig, da einer der folgenden Mnemonics fehlt: E, Z, NC, AE, etc.).

*~43 Parenthesis/Bracket Mismatch~*

Die Zahl der öffnenden und schließenden Klammern stimmt nicht überein.

*~44 Bad Forward Reference Combination~*

Der A86 kann die angegebene Vorwärtsreferenz nicht auflösen.

*~45 Is It Byte Or Word?~*

Es ist unklar, ob ein Byte oder Word adressiert werden soll. Beispiele sind fehlende Schlüsselworte BYTE oder WORD bei der indirekten Adressierung.

*~46 Bad #-Construct~*

Es wurde das #-Zeichen gefunden, ohne daß ein gültiger Makroparameter vorliegt.

*~47 #ENDIF Required~*

Zu dem #IF wurde keine korrespondierende #ENDIF-Anweisung gefunden.

*~48 #EM Required To End Macro~*

Das Makroende muß mit #EM abgeschlossen werden.

*~49 End Delimiter to COMMENT Required~*

Ein Kommentar wurde mit dem Schlüsselwort COMMENT begonnen und muß mit COMMENT beendet werden. Bis zum EOF wurde dieses zweite COMMENT nicht gefunden.

*~50 Reg,Mem Required~*

Eine ungültige Kombination von Operanden liegt vor. Tritt bei MOV, XCHG, ADD, SUB, CMP, XOR, etc. auf (z.B. der Versuch zwei Speichervariable zu addieren).

*~51 Segment Override Not Allowed Here~*

Eine Segment-Override Anweisung (ES: DS: CS) ist an dieser Stelle nicht erlaubt. Der Fehler tritt bei den impliziten Makros (z.B. MOV AX, BX) auf.

*~52 Byte Operand Required~*

Es wird ein Byte-Operand erwartet (z.B. bei STOBITS, LODBITS, ROL4, ROR4).

*~53 Word Register Required~*

Es wird ein WORD-Register für den Befehl oder Ausdruck erwartet (z.B. LDS, LES, LEA, BOUND, IMUL, LAR, LSL).

*~54 Floating-Point Chip Required~*

Ein Arithmetikprozessor ist nicht vorhanden, es wurden aber Fließkommaoperanden angegeben.

*~55 Bad Floating-Point Operand~*

Der angegebene Fließkommaoperand ist ungültig, d.h., der Speicheroperand kann eine undefinierte oder unkompatible Größe aufweisen, etc.

*~56 Constant 0--7 Required~*

Es wird eine Konstante zwischen 0 bis 7 erwartet. (Wird in Zusammenhang mit der 8087-Stackadressierung benutzt).

*~57 Memory Operand Required~*

An dieser Stelle wird ein Speicheroperand erwartet.

*~58 Segment Or Struc Name Not Allowed~*

Dieser Fehler tritt auf, falls eine für ein EXE-Programm geschriebene Quelle in eine COM-Datei übersetzt werden soll. COM-Programme erlauben keine Segmentangaben.

*~59 Word Operand Required~*

Es wird ein WORD-Operand benötigt. Der Fehler tritt bei 286-Befehlen auf (ARPL, SLDT, LLDT, STR, LTR, VERR, VERW, SMSW, LMSW).

*~97 Object Overflow~*

Der A86 hat einen Speicherüberlauf in seinem internen Objektcode Segment festgestellt. Die Vorwärtsreferenzen lassen sich nicht mehr auflösen. Der Fehler tritt nur auf, falls die Ausgabe die Grenze von 64 KByte (bei 200 KByte freiem Speicher) erreicht. Abhilfe schafft eine Aufteilung der Quelldatei in kleinere Files.

*~98 Undefined Symbol Not Allowed~*

Der Fehler tritt nur innerhalb des D86 im Line-Assembler-Mode auf, wenn ein Symbol angegeben wird, das nicht in der Tabelle vorliegt.

*~99 Symbol Table Overflow~*

Die Symboltabelle des A86 ist übergelaufen. Das Limit des A86 liegt bei mehreren 1000 Symbolen. Tritt der Fehler auf, muß die Zahl der Symbole im Programm reduziert werden. Eine andere Alternative ist die Aufteilung in kleinere Teilprogramme.

# Anhang E

## Attribute beim Monochromadapter

Bit 0-2: Vordergrundfarbe

Bit 3: Intensitätsbit

Bit 4-6: Hintergrundfarbe

Bit 7: Blink Bit

Darstellung	b7	b6	b5	b4	b3	b2	b1	b0
normal	b	0	0	0	i	1	1	1
invers	b	1	1	1	i	0	0	0
unterstrichen	b	0	0	0	i	0	0	1
weiß & weiß	b	0	0	0	i	0	0	0
schwarz & schw.	b	1	1	1	i	1	1	1

normal: weißes Zeichen auf schwarzem Hintergrund

invers: schwarze Zeichen auf weißem Hintergrund

b = 1 Zeichen blinkend

i = 1 erhöhte Intensität

# Anhang F

## Kodierung Farbattribute bei Colorkarten

Bit	Farbe
7	- Blinkbit
6	R Hintergrundfarbe
5	G Hintergrundfarbe
4	B Hintergrundfarbe
3	I Vordergrundfarbe
2	R Vordergrundfarbe
1	G Vordergrundfarbe
0	B Vordergrundfarbe

### **I R G B**

0 0 0 0 schwarz  
0 0 0 1 blau  
0 0 1 0 grün      Vorder- und  
0 0 1 1 cyanblau  
0 1 0 0 rot      Hintergrundfarbe  
0 1 0 1 violett  
0 1 1 0 braun  
0 1 1 1 weiß  
1 0 0 0 dunkelgrau  
1 0 0 1 hellblau  
1 0 1 0 hellgrün  
1 0 1 1 hellcyanblau  
1 1 0 0 hellrot    Vordergrundfarbe  
1 1 0 1 hellviolett  
1 1 1 0 gelb  
1 1 1 1 weiß

# Anhang G

## Die 8086-Assemblerbefehle

<b>Befehl</b>	<b>Name</b>
AAA	ASCII adjust AL for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiply
AAS	ASCII adjust for subtraction
ADC X1,X2	Add with carry
ADD X1,X2	Addition
AND X1,X2	Logical-AND
CALL NEAR	CALL im Segment
CALL FAR	CALL über Segmentgrenzen
CBW	Convert byte in word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt enable flag
CMC	Complement carry flag
CMP X1,X2	Compare
CMPS X1,X2	Compare Strings
CWD	Convert word to doubleword
DAA	Decimal adjust AL after addition
DAS	Decimal adjust AL after subtraction
DEC X1	Decrement
DIV X1	Unsigned divide
HLT	Halt
IDIV X1	Signed divide
IMUL X	Signed multiply
IN X1,X2	Input byte/word from port
INC X1	Increment X1 um 1
INT 3	Interrupt 3
INT xx	Interrupt xx
INTO	Interrupt on overflow
IRET	Interrupt return
JA cb	Jump short if above (CF=0 and ZF=0)
JAE cb	Jump short if above or equal (CF=0)
JB cb	Jump short if below (CF=1)
JBE cb	Jump short if below or equal (CF=1 or ZF=1)
JC cb	Jump short if carry (CF=1)

JCXZ cb	Jump short if CX register is zero
JE cb	Jump short if equal (ZF=1)
JG cb	Jump short if greater (ZF=0 and SF=OF)
JGE cb	Jump short if greater or equal (SF=OF)
JL cb	Jump short if less (SF/=OF)
JLE cb	Jump short if less or equal (ZF=1 or SF/=OF)
JMP cb	Jump short
JMP FAR	Jump far (4-byte address)
JMP NEAR	Jump near
JNA cb	Jump short if not above (CF=1 or ZF=1)
JNAE cb	Jump short if not above or equal (CF=1)
JNB cb	Jump short if not below (CF=0)
JNBE cb	Jump short if not below or equal (CF=0 and ZF=0)
JNC cb	Jump short if not carry (CF=0)
JNE cb	Jump short if not equal (ZF=0)
JNG cb	Jump short if not greater (ZF=1 or SF/=OF)
JNGE cb	Jump short if not greater or equal (SF/=OF)
JNL cb	Jump short if not less (SF=OF)
JNLE cb	Jump short if not less or equal (ZF=0 and SF=OF)
JNO cb	Jump short if not overflow (OF=0)
JNP cb	Jump short if not parity (PF=0)
JNS cb	Jump short if not sign (SF=0)
JNZ cb	Jump short if not zero (ZF=0)
JO cb	Jump short if overflow (OF=1)
JP cb	Jump short if parity (PF=1)
JPE cb	Jump short if parity even (PF=1)
JPO cb	Jump short if parity odd (PF=0)
JS cb	Jump short if sign (SF=1)
JZ cb	Jump short if zero (ZF=1)
LAHF	Load: AH = flags
LDS X1,X2	Load pointer using DS
LEA X1,X2	Load effectiv adress
LES X1,X2	Load pointer using ES
LODS	Load stringbyte
LODSW	Load streingword
LOOP XX	Loop DEC CX; jump short if CX/=0
LOOPE XX	Loop Equal DEC CX; jump short if CX/=0 and equal (ZF=1)
LOOPNE XX	Loop Not Equal DEC CX; jump short if CX/=0 and not equal
LOOPNZ XX	Loop Not Zero DEC CX; jump short if CX/=0 and ZF=0
LOOPZ XX	Loop Zero DEC CX; jump short if CX/=0 and zero (ZF=1)
MOV X1,X2	Move
MOVSB	Move Stringbyte
MOVSW	Move Stringword
MUL	Multiply unsigned
NEG	Negate
NOP	No Operation
NOT	NOT-Vergleich

---

OR X1,X2	Logical-OR
OUT X1,X2	Output to port
POP X1	POP Register from stack
POPF	Pop Flags
PUSH X1	PUSH Register to stack
PUSHF	PUSH Flags
RCL xx,1	Rotate left carry
RCR xx,1	Rotate 9right carry
REP (prefix)	Repeat
REPE (prefix)	Repeat equal
REPNE (prefix)	Repeat not equal
REPZ (prfix)	Repeat not zero
REPZ (prefix)	Repeat zero
RETF	Return FAR
RET	Return NEAR
ROL xx,1	Rotate left
ROR xx,1	Rotate right
SAHF	Store AH into flags
SAL xx,1	Shift arithmetik left
SAR xx,1	Shift arithmetik right
SBB X1,X2	Subtract with borrow
SCAS	Compare Strings
SHL xx,1	Shift left
SHR xx,1	Shift right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt enable flag
STOS xx	Store String
SUB X1,X2	Subtract
TEST X1,X2	Test
XCHG X1,X2	Exchange
XLAT xx	Translate
XOR X1,X2	Exclusive-OR

# Literatur

/1/ Born Günter: MS-DOS 6.2 Handbuch zur Programmierung, Microsoft Press, München, 1994

/2/ Born Günter: DOS 6 Tuning, Tips, Tricks und Utilities zur effektiven Batch-Programmierung. Markt&Technik Verlag, 1993, München

/3/ Monadjemi, P.: PC Programmierung in Maschinensprache, Markt & Technik Verlag, 1990.

/4/ N.N.: iAPX 86,88 User Manual, Intel, 1982



# Stichwortverzeichnis

<b>!</b>		Universalregister.....	28
!-Operator.....	262	8087.....	257
<b>#</b>		<b>A</b>	
#ELSIF Beding.....	291	A86	
#EM.....	285; 290	#EM.....	284; 285
#ENDIF.....	290; 291	\$ .....	264
#EX.....	285	'>'.....	263
#IF Beding.....	291	= .....	272
#Nn-Option.....	290	>-Zeichen.....	283
#Sn-Option.....	289	Advance Option.....	288
#V-Option.....	289	After.....	288
<b>+</b>		align.....	276
+V-Schalter .....	405	Anweisungen zur Segmentierung.....	265
<b>.</b>		Assemblierung.....	291
.SYM.....	405	Auswertung eines Ausdrucks.....	265
<b>3</b>		bedingte Assemblierung.....	279
32-Bit-Adreßvektor.....	243	Before.....	288
3-Operanden-Befehl.....	254	BIN-Datei.....	291
<b>4</b>		CODE ENDS.....	265
4-Byte-Rückkehradresse.....	243	CODE SEGMENT.....	265
4-Byte-Vektor.....	244	COM-Datei.....	291
4CH .....	244	DATA SEGMENT.....	266
<b>8</b>		DB.....	266; 267
80186/80286-Prozessor.....	253	DD.....	267
80287.....	257	DQ.....	267
8080-Mode.....	251	DW .....	266; 267
8086		EQU \$ .....	265
Befehlsgruppen.....	39	EXE-Datei.....	291
Flags.....	31	explizites Makro.....	283
Kodierung Flags.....	31	Generierung der Segmente.....	265
Prozessor.....	27	Konstanten.....	259
Registerstruktur.....	28; 152	Linken.....	291
		lokales Label.....	263
		Makro-Aufruf.....	284
		Makrodefinition.....	285
		Makros.....	279
		Parameter #1.....	284
		STRUC.....	266

TB.....	267
A86-Directive.....	259
A86-Option.....	293
AAA.....	96; 215
AAD.....	223
AAM.....	220
AAS.....	99; 218
absoluter Sprung.....	230
ADC.....	93; 212
ADD.....	92; 211
Add with Carry.....	93; 212
ADD4S.....	251
Addition.....	261
Addition von gepackten BCD- Zahler	251
Adresse der Speicherstelle.....	169
Adresse der Variablen.....	264
Adresse des Vektors.....	244
Adressierung der Strings.....	248
Adreßkonstante.....	264
Akkumulator (AX).....	28; 153
ALT-F10.....	405
AND.....	73; 193; 262
Funktion.....	24
Anfangsadresse des Unterprogramme	242
Arithmetik Befehle.....	89; 208
ARPL.....	253
ASCII	
Tabelle.....	68
ASCII-Adjust for Addition.....	96; 215
ASCII-Adjust for Division.....	104; 223
ASCII-Adjust for Subtraktion.....	99; 218
ASCII-HEX-Konvertierung.....	298
ASK.ASM.....	237; 240
ASK.OBJ.....	296
Assembleranweisungen.....	21
Auflistung von Symbolen.....	274
Ausgabemodule.....	300
Autoincrement/-decrement-Funktion	248
Auxillary-Carry-Flag.....	31; 155
Auxillary-Flag.....	74; 193
<b>B</b>	
Base Pointer (BP).....	30
Base Pointer Register (BP).....	154
Base Register (BX).....	153
Base-Register (BX).....	29
BASIC.....	21
Basis Adressierung.....	53
Basis Index Adressierung.....	53
BCD-Zahl.....	91; 210
Bearbeitung von Strings.....	248
Bedingte Assemblierung.....	290
Bedingte RETURN-Anweisung.....	281
bedingte Sprungbefehle.....	122; 232
Befehl	
AAA.....	96
AAD.....	104
AAM.....	102
AAS.....	99
ADC.....	93
ADD.....	92
AND.....	73
CALL FAR.....	138
CALL NEAR.....	135
CBW.....	109
CMP.....	105
CWD.....	109
DAAL.....	95
DAS.....	99
DEC.....	107
DIV.....	102
ESC.....	149
HLT.....	149
IDIV.....	103
Immediate MOV.....	42
IMUL.....	101
INC.....	106
INT.....	141
IRET.....	142
JMP FAR.....	119
JMP NEAR.....	116
JMP SHORT.....	117
LAHF.....	72; 191
LDS.....	70; 189
LEA.....	69
LES.....	70; 189
LOCK.....	149
LOOP.....	145
LOOPE/LOOPZ.....	145
LOOPNE/LOOPNZ.....	146
MOV.....	39
MUL.....	99
NEG.....	108
NOP.....	67

NOT.....	73
OR.....	76
OUT.....	63
POP.....	56
POPF.....	58
PUSH.....	54
RCL.....	86
RCR.....	87
ROL.....	84
ROR.....	86
SAHF.....	72; 191
SAL.....	81
SAR.....	83; 202
SHL.....	81; 200
SHR.....	82; 201
TEST.....	79
WAIT.....	149
XCHG.....	64
XLAT.....	67
XOR.....	77
Befehl JNO.....	240
Befehl JNP/JPQ.....	240
Befehl JNS.....	241
Befehl JO.....	241
Befehl JP/JPE.....	241
Befehl JS.....	241
Befehl LEA.....	188
Befehl SUB.....	96
Befehle	
zur Bitmanipulation.....	73
Befehle zur Bitmanipulation.....	192
Binärwerte.....	22
Binärzahlen.....	259; 305; 343
BIN-Programme.....	266
BIOS-Datentabelle.....	60; 181
BIOS-ROM.....	232
BIT.....	261
Bitfeld.....	251
B-Operator.....	288
BOUND.....	253
BY-Operator.....	260
BYTE.....	263
<b>C</b>	
C.....	302
CALL.....	238; 242
CALL Befehle.....	135
CALL FAR.....	138; 243
CALL NEAR.....	135; 242
CALL80 imm8.....	251
Carry-Flag (CF).....	31; 155
CBW.....	227
class name.....	277
Clear Carry-Flag (CLC).....	87; 206
Clear Direction-Flag (CLD).....	88; 207
Clear Interrupt-Enable-Flag (CLI).....	88; 207
Clear Task Switched Flag.....	253
Clipper.....	302
C-LOOP.....	287; 288
CLRBIT Op1,Op2.....	251
CLTS.....	253
CMP.....	223
CMP4S.....	251
CMPS.....	148
CMPS Anweisung.....	148
CMPS-Anweisung (Compare String).....	249
CODE SEGMENT.....	268
CODE SEGMENT-Directive.....	300
CODE,DATA und STACK Direktive.....	278
Codesegment Register (CS).....	160
combine.....	277
COM-Datei.....	168; 240; 266
compare while not end of string.....	249
Complement Carry-Flag (CMC).....	88; 207
COM-Programme.....	266
Convert Word to Double Word.....	109; 228
Count Register (CX).....	153
Count-Register (CX).....	29
CWD.....	228
CX-Register.....	248
<b>D</b>	
DAA.....	95; 213
DAS.....	99; 217
DATA ENDS.....	266
Daten Register (DX).....	153
Datenbereiche.....	306
Datendefinition.....	267
Datenformate.....	89; 208
Daten-Register (DX).....	29
Datensegment.....	265; 266
Datensegment Register (DS).....	160
Datenstruktur.....	269
DB.....	306; 344

DB-Anweisung.....	168; 268	DWORD.....	243; 263
DD.....	306; 344	<b>E</b>	
DEBUG.....	377	Effective Address Operanden.....	255
ASSEMBLE.....	390	Einführung	
COMPARE.....	385	8086-Befehlssatz.....	152
DUMP.....	379	ELSE-Anweisung.....	291
ENTER.....	380	ELSEIF-Teil.....	290
File I/O-Befehle.....	387	END.....	332
FILL.....	382	END-Directive.....	276
GO.....	397	ENTER.....	254
HEX.....	384	Environment.....	295
INPUT-/OUTPUT.....	383	EQU-Directive.....	270
MOVE.....	383	Errorcode.....	238
PROCEED.....	398	ERRORLEVEL-Funktion.....	244
Programmentwicklung.....	389	Erweiterung DOS-Befehlssatz.....	237
REGISTER.....	386	Erzeugung von BIN-Dateien.....	293
SEARCH.....	385	Erzeugung von COM-Dateien.....	292
TRACE.....	398	Erzeugung von OBJ-Dateien.....	293
UNASSEMBLE.....	395	ESC.....	250
Debugger D86.....	405	ESC.EXE.....	296
DEC.....	225	EVEN-Directive.....	267
Definition von Strukturen.....	269	EXE-Programm.....	293
Demonstration des INT-Befehls.....	245	explizites Makro.....	279
Destination Index (DI).....	29; 154	Extended ASCII-Code.....	238
Dezimal Adjust for Addition.....	95; 214	Extended MOV- und XCHG- Anweisungen.....	282
Dezimalwerte.....	305; 343	externer Prozessor.....	250
Dezimalzahl.....	259	Extrasegment Register (ES).....	160
Dezimalzahlen.....	22	EXTRN.....	332
Direction Flag.....	255	EXTRN-Directive.....	274
Direction-Flag (DF).....	32; 156	<b>F</b>	
Direction-Flags.....	248	FAR Pointer.....	269
Directiven.....	259	Fehlermeldungen des A86.....	409
Direkte Adressierung.....	52; 242	Feldindex.....	253
Displacement.....	113; 230	Flagregiser.....	244
DIV.....	221	Flags.....	31; 155
Division.....	261	Kontrolle.....	87
DOS-Aufruf INT 21, AH = 4CH.....	168	Fließkommabefehl.....	257
DOS-EXIT.....	61; 182	FORTTRAN Bereich.....	277
DOS-INT 21 Funktion AH = 02.....	238	Fragezeichen.....	268
DOS-Routine per INT 21.....	244	Frame.....	254
DOS-Versionsabfrage.....	291	<b>G</b>	
DQ.....	306; 344	gepackte Darstellung.....	91; 209
Druckerports.....	61; 182	Global Descriptor Table.....	256
DT.....	306; 344		
DUP.....	268		
DUP ?.....	268		
DW.....	306; 344		

Global Descriptor Table Register.....	255	INT 3.....	142; 244
Größe von Variablen.....	263	INT 5.....	253
GROUP.....	333	INT O.....	142
GROUP-Deklaration.....	277	Interrupt Descriptor Tabelle.....	255
GROUP-Directive.....	278	Interrupt Descriptor Table.....	256
<b>H</b>		Interrupt-Controller.....	244
HALLO.ASM.....	167	Interrupt-Flag (IF).....	32; 156
HALT-Mode.....	250	Interrupt-Nummer.....	244
Hardwareinterrupt.....	244; 250	Interrupt-Service-Routine.....	244
Hardwareunterbrechung.....	244	Interrupt-Service-Routinen.....	142
Hexadezimalsystem.....	22	INTO.....	244
Hexadezimalwert.....	305; 343	IP-Register.....	230
Hexadezimalzahl.....	259	IRET.....	142; 245
HEXASC.ASM.....	297	<b>J</b>	
HIGH/LOW-Operator.....	260	JA /JNBE.....	123; 233
HLT.....	149; 250	JAE /JNB.....	124; 234
<b>I</b>		JB /JNAE / JC.....	125; 235
IDIV.....	222	JBE /JNA.....	125; 235
IF-Bedingung.....	290	JCXZ.....	125; 235; 246
IF-Statement.....	280	JE / JZ.....	126; 236
Immediate-MOV.....	166	JG /JNLE.....	128; 236
Immediate-MOV-Befehl.....	42	JGE /JNL.....	128; 236
implizites Makro des A86.....	279	JL /JNGE.....	129; 236
IMUL.....	219; 254	JLE /JNG.....	129; 237
IN.....	62; 183	JMP.....	242
INBefehl.....	62	JMP Befehle.....	110; 228
INC.....	224	JMP FAR.....	119; 231
Index Adressierung.....	53	JMP NEAR.....	111; 116; 229; 270
Index Register.....	154	JMP SHORT.....	111; 117; 230; 270
Indirekte Adressierung.....	168; 242	JNC.....	129; 237
beim MOV-Befehl.....	46	JNE /JNZ.....	129; 237
Initialisierung der Daten mit 0.....	268	JNO.....	132
Initialisierung von kompletten		JNP / JPO.....	133
Datenbereichen.....	249	JNS.....	133
INSB (Input String Byte).....	254	JO.....	133
Instruction Pointer (IP).....	30; 155	JP / JPE.....	133
Instruktionspointer.....	242	JS.....	134
Instruktionssatz		Jump if No Parity/Jump if Parity Odd.....	240
erweiterter.....	286	Jump if Overflow.....	241
INSW (Input String Word).....	254	Jump if Sign.....	241
INT.....	141; 244	Jump No Sign.....	241
INT 21.....	61; 182	Jump Not Overflow.....	240
INT 21 Funktion AH = 4CH (DOS-Exit).....	238	Jump on Parity/Jump if Parity Even.....	241
INT 21-Funktion.....	244	<b>K</b>	
INT 21-Funktion AH = 08H.....	238	Kodierung	

8086-Flags.....	155	LOOPNE (Loop While Not Equal).....	247
Kommandozeile.....	130; 238	LOOPNE/LOOPNZ.....	247
Konstante.....	229; 305	LOOPNZ (Loop While Not Zero).....	247
Konstruktion von Schleifen.....	246	LOOPZ (Loop While Zero).....	247
Kontrolle		L-Operator.....	287
Flags.....	206	LPT1.....	60; 181
Kontrolle an das Betriebssystem.....	244	LPTSWAP.....	61; 182
<b>L</b>		LSL.....	255
LABEL FAR.....	273	LTR.....	255
LABEL-Directive.....	273	<b>M</b>	
Lage des Datensegmentes.....	266	MACRO.....	285
LAHF.....	72	MAIN-Directive.....	275
Länge des Strings.....	289	Makrobefehl.....	285
Länge einer Datenstruktur.....	264	Marke.....	229
LAR.....	254	Marke für einen Sprung.....	273
LDS.....	70	Maschine Status Word.....	255
LEA.....	69	Maschinenprogramm.....	21
LEA-Instruktion.....	283	MASM	
LEAVE.....	254	DUP.....	307
LES.....	70	EQU Directive.....	315
LGDT.....	255	EVEN-Directive.....	319
LIDT.....	255	EXE-Datei.....	324
LINK.EXE.....	22	LENGTHOF.....	309
Linken von OBJ-Dateien.....	295	OFFSET Operator.....	312
Linker.....	274; 302	Operationen auf Ausdrücken.....	316
Link-Option.....	296	ORG-Anweisung.....	312
Liste der Fehlermeldungen.....	292	PROC Directive.....	323
LLDT.....	255	Segmentanweisung.....	311
LMSW.....	255	SIZEOF.....	309
Load Access Right Byte.....	254	Strukturen.....	310
Load AH-Register from Flags.....	72; 191	TYPE.....	310
Load Data Segment.....	71; 190	Mehrfach PUSH-, POP-, INC- und DEC-	
Load Extra Segment.....	71; 190	Befehle.....	280
Load Task Register.....	255	Mehrfachdeklaration mit EQU.....	272
Local Descriptor Table Register.....	255	Menüoberfläche in DOS.....	240
LODBIT Op1, Op2.....	251	MODULE.ASM.....	297
LODS.....	148	MOV.....	163
LODS Anweisung.....	148	MOV Befehl.....	39
LODS-Anweisung (Load String).....	249	MOVS.....	147
logische Operatoren.....	262	MOVS Anweisungen.....	147
Lokales Label.....	274; 282	MOVS-Anweisung (Move-String).....	248
Lokales Symbol.....	274	MUL.....	99; 218
LONG.....	263	Multiplikant.....	254
LOOP.....	246; 252	Multiplikation.....	261
LOOPE (Loop While Equal).....	247		
LOOPE/LOOPZ.....	247		

**N**

NAME-Directive.....	273
NEAR.....	263
NEAR-Sprung.....	263
NEC-Befehlssatz.....	250
NEG.....	227
Negative Schleifen.....	288
NEG-Operator.....	262
Nibble.....	91; 210
NIL.....	271
NOP.....	67; 187
NOP-Anweisung.....	267
NOT.....	73; 192; 262
Funktion.....	25
NOTBIT Op1,Op2.....	252

**O**

OBJ-Datei.....	302
OBJ-Dateien.....	266
OFFSET.....	168; 263; 264
oktale Zahlen.....	259
Oktalzahlen.....	305; 343
Operation auf Ausdrücken.....	260
Optimierung durch den A86-Assembler.....	283
OR.....	76; 195; 262
Funktion.....	24
ORG-Anweisung.....	266
ORG-Directive.....	266
OUT.....	63; 184
OUTSB (Output String Byte).....	255
OUTSW (Output String Word).....	255
Overflow-Flag (OF).....	31; 156

**P**

Paragrafen.....	270
Parameterstring.....	238; 287
Parameterübergabe.....	302
Parity-Flag (PF).....	31; 156
Patch-Mode.....	406
POP.....	56; 178
POPA.....	255
POPF.....	179
Portadresse.....	63; 184
Postfix.....	259
Prefix.....	250
Prioritäten.....	265
Privileg Ebene 0.....	253

PROC-Directive.....	272
Programm-Segment-Prefix.....	130; 238
Programmstartadresse.....	275
Programmtransfer Befehle.....	110; 228
Programmunterbrechung.....	244
Protected Mode.....	253
Prozedur.....	272
Pseudobefehl.....	259
PTR.....	261
PUBLIC.....	331
PUBLIC-Anweisung.....	283
PUBLIC-Directive.....	274
PUSH.....	54; 176
PUSH imm.....	256
PUSHA.....	177; 256
PUSHF.....	56; 177

**Q**

QS als Override.....	271
Quelldatei.....	21
QuickBasic.....	302
QWORD.....	263

**R**

RADIX 16.....	259
RADIX-Directive.....	305
RADIX-Kommando.....	259
RCL.....	86; 205
RCL/RCR/ROL/ROR/SAL/SAR/SHL/SHR .....	256
RCR.....	87; 206
Real Mode.....	27; 32; 152; 157
REAL-Mode.....	253
Register	
Codesegment (CS).....	35
Datensegment (DS).....	35
Extrasegment (ES).....	36
Stacksegment (SS).....	36
Register-indirekte Adressierung.....	52
REP-Anweisung.....	255
REPC/REPNC.....	252
REPE/REPNE/REPZ/ REPNZ.....	249
REPE/REPZ-Anweisung.....	249
REPEAT.....	248
REPEAT Anweisungen.....	147
REPEAT-UNTIL-Schleife.....	247
Reset.....	250

RESET.ASM.....	232
RET .....	139; 243; 272
RET 2.....	141; 244
RETF.....	243; 245
RETF 2.....	142
RETF-Anweisung.....	272
R-LOOP.....	286; 287; 288
ROL.....	84; 203
ROL4 Op1.....	252
ROR.....	86; 204
ROR4 Op1.....	252
Rotate Befehle.....	84
Rotate trough Carry Right.....	87; 206
Rotate-Befehle.....	203
RPL Feld.....	253
Rückkehradresse.....	243
Rücksprungadresse.....	244
<b>S</b>	
SAHF.....	72
SAL.....	81; 200
SAR.....	83
SBB.....	97; 216
Befehl.....	97
SCAS.....	148
SCAS Anweisung.....	148
SCAS-Anweisung (Scan String).....	249
Schachtelung von Schleifen.....	288
Schalter .....	293
Schiebeoperatoren.....	261
Schleifen.....	125; 144; 235
Schleifen in Makros.....	286
SEG .....	312; 349
SEG-Directive .....	279
Segment	
Offset.....	33; 158
Segment Limit.....	255
Segmentadresse.....	276; 312
SEGMENT-Anweisung.....	278
SEGMENT-Directive.....	276
Segment-Override.....	172; 242
Segment-Override-Befehl.....	53
Segment-Override-Befehlen.....	271
Segment-Overrides.....	285
Segmentregister.....	32; 157
Selektor .....	253
Semaphore	
mit XCHG.....	66
Set Carry-Flag (STC).....	88; 207
Set Direction-Flag (STD).....	88; 207
Set Interrupt-Enable-Flag (STI).....	89; 208
SETBIT Op1,Op2.....	252
SGDT.....	256
Shift Arithmetic Right.....	83; 202
Shift Befehle .....	80; 199
Shift Logical Right.....	82; 201
SHL.....	81
SHORT.....	263
SHORT-Sprung.....	111
SHR.....	82
SHR, SHL .....	261
SIDT.....	256
Sign-Flag (SF).....	31; 156
Single-Step-Mode.....	271
Software-Interrupt.....	244
Source Index (SI) .....	29; 154
Speicherbereiche.....	249
Speicherbereiche vergleichen.....	249
Sprung über die Segmentgrenzen.....	231
Sprungziel.....	112; 229
Stack Pointer (SP) .....	30; 154
Stackbereich.....	277
Stackframe.....	254
Stacksegment.....	170
Stacksegment Register (SS) .....	160
Standardvorgabe.....	296
Steueranweisungen.....	21
Steuerung des Übersetzervorgangs..	259
STOBITS Op1, Op2.....	252
Store AH-Register to Flags (SAHF)72; 192	
STOS.....	148
STOS Anweisung.....	148
STOS-Anweisung (Store String).....	249
STR .....	256
String Befehl.....	248
String Befehle.....	147
Stringvergleich .....	263
STRUC.....	269
SUB.....	96; 215
SUB4S.....	252
Subtract with Borrow.....	97; 216
Subtraktion	
gepackter BCD-Zahlen.....	252
Subtraktion von Konstanten.....	261

SWAP.ASM.....	187	Übergabe von Werten.....	289
Systemroutine.....	244	Überlauf.....	241
<b>T</b>			
Task Register.....	256	<b>U</b>	
TASM			
\$-Symbol.....	353	ungepackte Darstellung.....	90
Datenbereiche.....	344	Universalregister.....	28; 152
DUP.....	345	Unterprogramme.....	242
END.....	370	<b>V</b>	
EQU Directive.....	352	V20/V30 Prozessor.....	250
EVEN Directive.....	356	Variable in Strukturen.....	274
EXTRN.....	369	Vektortabelle.....	244
GROUP.....	370	Vergleichsoperator.....	262
Konstante.....	343	Verify Segment for Read (VERR).....	256
LENGTHOF.....	346	Verify Segment for Write (VERW)....	256
logische Operatoren.....	355	VERR/VERW.....	256
MODEL-Directive.....	347	Virtueller Debug-Mode.....	405
OFFSET Operator.....	349	Vorwärtsreferenz.....	266; 269
Operationen auf Ausdrücken.....	354	<b>W</b>	
ORG-Anweisung.....	349	WAIT.....	149; 250
PROC Directive.....	360	Warmstart.....	232
PUBLIC.....	369	Wert	
RADIX-Directive.....	343	aktueller Programmzeiger.....	264
Segmentadresse.....	349	while strings are equal.....	249
Strukturen.....	347	WORD.....	263
TEST.....	79; 198	<b>X</b>	
TESTBIT Op1,Op2.....	253	XCHG.....	64; 185; 250
TEST-Operation.....	283	XLAT.....	67; 188
Textkonstante.....	268	XOR.....	77; 197; 262
THIS-Operator.....	264	Funktion.....	24
TOP_OF_STACK.....	277	<b>Z</b>	
TRAP.....	271	Zeichen @.....	296
Trap-Flag (TF).....	32; 157	Zeigerregister.....	269
Turbo Pascal.....	302	Zero-Flag (ZF).....	32; 156
TWORD.....	263	Zugriff auf gerade Adressen.....	267
Typ der Referenz.....	275	Zugriffsprivileg.....	253
TYPE-Operator.....	264	Zweierkomplement.....	31
<b>Ü</b>			
Übergabe der Operandengröße.....	289		